UNIVERSITY OF OSLO

Department of informatics

Agile Scientists? Investigating Agile Practices in Scientific Software Development

Master thesis

60 credits

Magnus Thorstein Sletholt

August 2011

Acknowledgements

First and foremost, I would like to thank my supervisors at Simula, Jo Hannay and Hans Petter Langtangen. Their accessibility, patience and willingness to discuss the thesis have been of great help. I am grateful for the level of freedom given to me when designing and conducting the study. I really appreciate the vote of confidence. Also, I owe me deepest gratitude to Dietmar Pfahl, for numerous contributions. His quick and detailed feedback has been a tremendous support throughout the writing process.

I would like to thank the University of Oslo and Simula for creating excellent educational and working environments, and for giving me the chance to write this thesis.

I am thankful to Hans Christian Benestad for his help with the construction of the agile mapping chart.

I would like to express my gratitude to the FEniCS, Dalton and Olga projects, in particular the interviewees, who participated in the case study.

Last, but not least, I would like to thank my girlfriend, Elizabeth Kirknes, who has supported me and even assisted with proofreading of the thesis.

Abstract

The topic of this master thesis is development of scientific software. The research questions put forth are oriented towards specific agile practices and whether these are present in the development processes of scientific software projects. Moreover, the effects of applying such agile practices, particularly pertaining to the handling of requirements and testing, in scientific software projects are addressed in the thesis. In order to answer the proposed research questions a table consisting of 35 agile practices associated with two central methodologies, Scrum and Extreme Programming, have been applied.

Two research methodologies have been used in this thesis; a systematic literature review and a case study. The literature review has identified projects reported in scientific articles, where agile practices, both intentional and unintentional, have been observed. These projects have been appraised in detail to determine which practices were used and what effects, if any, these practices had. Three well-established development projects from different scientific domains, FEniCS, Dalton and Olga, have been investigated in a multiple case study. In each of these projects, 2 – 4 key developers were interviewed in semi-structured interview sessions, consisting of one part focusing on the overall development process and a second part concerning the 35 agile practices.

As to the presence of the agile practices in the projects examined, there were certain practices that appeared to be very popular and widely used. Some practices were difficult to evaluate (especially for the projects examined in the systematic literature review), while others were rarely applied. There were some differences among the projects and also some major differences between the projects in the case study and projects in the systematic literature review, in terms of which practices actually were used. The observed effects of agile practices have been promising, especially for testing. Due to the characteristics of the projects examined, and due to the size of the sample, further research must be conducted in order to obtain conclusive answers with regards to the use of agile practices in scientific software and the effects thereof.

Table of contents

List of tables

List of figures

1 Introduction

1.1 About the thesis

This thesis consists of two main research parts. The first part is a systematic literature review examining the use of agile methods and practices in scientific software projects. The second part is a case study investigating the development processes in a selection of scientific software projects, both on a general level and from an agile-specific viewpoint.

The first chapters focus on prior research into general scientific software development, the different research methodologies used in the study, as well as on the assessment of agility measurements. An overview of scientific software development in general, along with common challenges in such development, will hopefully be obtained in the part concerning previous research. The research methodologies of systematic review and case study will then be presented respectively. Thereafter, various agility measurement techniques and practices will be reviewed in order to construct an agile mapping chart – a tool for assessing the agility in the projects under investigation in both the systematic literature review and the case study.

The construction of the agile mapping chart leads up to the first main part of the thesis: the systematic literature review on agile practices and their effects in scientific software development. The effects assessed are primarily related to requirements and testing activities, which were identified as challenging in the initial literature appraisal, found in chapter 2, of scientific software development.

Following the literature review, the case study is presented. The case study primarily focuses on the development processes in a selection of scientific software projects, a total of three projects have been examined. 2 – 4 key developers from each project have been interviewed in order to obtain their perception of the development. The agile mapping chart will be used to determine which agile practices are present in the projects. The effects mentioned in the literature review will be investigated if related agile practices were in fact applied. In the final stages of the case study, a comparison will be made, summing up the general trends and results.

Consequently, the results from the case study and the results from the systematic literature review will be discussed in relation to the two research questions. Conclusions and answers to the research questions, a presentation of the inherent limitations (including validity threats) of the study and suggestions to further studies are found in the final parts of the thesis.

1.2 Research questions

The nature of scientific research and the development of scientific software have certain similarities with processes following agile methods: responsiveness to change and collaboration are of the utmost importance. Of course, before recommending any scientists to introduce agile methods in their projects, further investigations are needed. Firstly, thorough examination of the actual processes in scientific software must be conducted to obtain key characteristics of this type of projects. Secondly, the effects of using agile practices must be assessed, with particular emphasis on how these practices affect perceived key challenges in scientific software projects. Thirdly, investigations into which agile practices already or frequently present in representative scientific software development projects must also be mapped out.

Fortunately, a number of recent studies describe the general lines of scientific software and the regular development practices found therein. By reviewing these studies (as done in chapter 2), a set of aspects forming the characteristics of such development processes, as well as associated difficulties and challenges, may be extracted.

Two main research questions may thus be formalized as follows:

- RQ1: *How well do practices in current scientific software development processes match the practices found in agile development methods?*
- RQ2: *How does the use of agile practices influence the handling of commonly regarded challenging aspects in scientific software development projects?*

RQ2 focuses on testing and requirements activities, identified as challenging aspects in the review of prior research (presented and elaborated in chapter 2). The two following propositions are investigated in relation to research question 2:

> **P1**. Projects using agile practices have a better handling of testing-related activities.
>
> **P2**. Projects using agile practices have a better handling of requirements activities.

In the outset, it is unknown how many agile practices are in fact present in the projects investigated in the case study. If these projects do not apply practices related to testing and requirements, they are of limited relevance to research question 2 and its two propositions. There are also uncertainties for the systematic review, as there might be insufficient information available in the identified studies about which specific agile practices that were applied. However, both research questions will be addressed to the extent possible in each study.

1.3 Motivation

The motivation for addressing research questions RQ1 and RQ2 is multilateral. The ultimate goals in this line of research are to further more explicit and deliberate scientific software practices and evaluate how to find potential solutions to pressing issues in scientific software projects, preferably by utilizing already established and proven concepts from software engineering. By working towards these goals, an update of software engineering to include scientific software domains is also achieved. To believe that these ambitions could be fulfilled by this thesis alone is certainly too optimistic, but the thesis may at least contribute to the research attempting to bridge the perceived “chasm” [1] between scientific computing and software engineering.

There are also other reasons for carrying out this study. One is to continue research on scientific software development, in which iterative development and agile tendencies have been observed in multiple studies [2; 3; 4]. The actual application of specific agile practices in scientific software has, however, not yet been addressed directly. It has also been put forth that agile development models might suit the needs and nature of scientific software projects very well [5]. Hopefully, the cases in the case study aligns well with the projects examined in previous research, making the results in this thesis generally applicable and valid to the extent where one is able to identify correlations between them.

Prior research has indicated some areas or aspects of development which have been reported to be quite difficult to handle by scientists developing software. This study explores one potential way (using agile practices) of improving those aspects of the development processes. It will be interesting to see whether the use of agile methodologies may to some extent resolve such issues. Of course, unintended side effects of an agile approach must also be assessed in order to obtain a good evaluation of the suitability of such methods. The specific challenges I will emphasize are described in chapter 2, where the general trends of earlier research into the development of scientific software are summarized.

1.4 Definitions

This section contains brief definitions and descriptions of *scientific software* and *agile software development*. The purpose of including these definitions is to precisely delineate the meaning of these terms in the remainder of the thesis, as the definitions of some terms may vary, especially the agile methodologies *Scrum* and *Extreme Programming (XP)*, depending on which source is used. It is especially important to have a clear reference model for the agile methodologies, as these are used extensively throughout the thesis.

The first subsection describes the nature of scientific software, and how key aspects of such development diverge from those of regular software development contexts. Next, agile software development is described, followed by presentations of two renowned agile methodologies, Scrum [6] and XP [7].

1.4.1 Scientific software

The aim of scientific software is to grasp or solve a scientific problem. Such software incorporates a significant scientific component and is often found in domains related to mathematics and natural sciences. These pieces of software are vital tools for simulating, conceptualizing, analyzing and visualizing scientific data and natural phenomena. Scientific software is also very important for investigating scientific theories. This means that scientific software and the development of such software is an important part of research, especially in mathematics and other natural sciences.

The developers involved in such projects are mostly scientists, not professional software engineers. Although basic introduction in computer science, at least some programming training, is usually provided in the curricula of all natural science degrees, very few of the scientists are familiar with the theoretical, managerial or organizational aspects of software development. Due to the complexity of scientific software, it can be hard to find people who are able to contribute to the project (apart from highly educated scientists), as it may require years of education, training and research experience just to comprehend the scientific domain, not to mention the specifics of the tasks.

The development of scientific software is not exclusively performed by individuals working fulltime in a project, despite large projects with a life-span of several years. Scientists involved in software development projects are also otherwise engaged in research, education or even other scientific software projects. Full-time commitment to a specific software project is rare. However, scientists use an increasing part of their professional work time developing applications [8], thus making the development of such software an apt area of research.

The drivers for change are also somewhat different in scientific software projects than in commercial software development projects (with members having full-time commitment to the project). The need for modifications is motivated by scientists' individual preference, as well as by changes in the underlying research or the scope of the implemented research. There are also other factors which may warrant modifications, such as accuracy or performance demands or portability concerns.

1.4.2 Agile methodologies

Agile methodologies emerged in the mid 1990s as an alternative to the traditional, plan-driven approach to software development. Judging by the reception and current spread of these methodologies, the light-weight processes prescribed by these methodologies fulfilled a latent need in the general software development community. Early versions of well-known agile methods, like Scrum and XP, were introduced to the software engineering scene in 1995 and 1996 respectively.

In 2001, a group of software engineers formulated the agile manifesto [9], in which profound principles of agile development are described. The four main points are:

(1) "Individuals and interactions over processes and tools"
(2) "Working software over comprehensive documentation"
(3) "Customer collaboration over contract negotiation"
(4) "Responding to change over following a plan"

Frequent releases are a common feature in agile methods; the development is divided into monthly or semi-monthly iterations. In these iterations the programmers implement code additions or improvements. Iterations culminate with a release of the newly implemented software. The new functionality is often presented to customers or users at a special review meeting, where anyone interested are welcome to attend. Prior to a release, most of the functionality should have been subject to some kind of quality assurance or testing, with regards to both code and user acceptance. There are a lot of factors to consider when selecting the tasks to be solved in the next iteration, but the general guideline is to choose the tasks generating the highest business-value.

Agile development projects aspire to be concentrated around highly motivated individuals. The developers should feel a strong ownership of the code and actively communicate with users/customers. In that respect, the teams are to a high degree autonomous and self-organized. There are often predefined roles for the team members, describing their objectives and responsibilities in the development. The teams ought to be cross-functional, meaning that the members within a team have to complement each other's skills and personality.

Agile methods/methodologies are founded on an enduring commitment, not only from developers; users and customers are also encouraged to actively participate in certain activities during the short iterations. To be able to respond to change, the projects need constant input and feedback from their users – both to define and delineate requirements, and to maintain and refine the already existing functionality. Contrary to more plan-based methods, with distinct phases, activities and transitions between these, agile methods have interwoven all such activities into monthly or semimonthly iterations.

1.4.3 Scrum

Scrum [6] is a prescriptive process model that defines roles in a development project, as well as the activities to be performed in the *iterations* or, as they are referred to in Scrum jargon, *sprints.*

In Scrum, there is a predefined set of roles for the team members. The most important roles are the *Scrum Master*, the *Product Owner* and regular team members. The Scrum Master is a project facilitator whose primary objective is to keep the team's development *velocity* on a satisfactory level, and is the closest to a project leader in the traditional sense. The Product Owner ensures that sufficient resources and information about the tasks are available. Scrum primarily focuses on practices and organization within a team, yet it has also been proven to adapt well to larger development projects with multiple teams and distributed development [10]. "Scrum of Scrums" is an alternative way to go in such cases.

During the time-boxed sprints the development team performs requirement planning, task estimation and formalization, in addition to coding and review activities. In the initial stages of a sprint, a sprint planning meeting is arranged, where the set of prioritized tasks are the matter of discussion; during this meeting tasks are broken down (if necessary) and estimated, and eventual ambiguities are sorted out by the Scrum team. *Planning poker*, described in greater detail in section 4.5.1, is often used to estimate tasks. Most of the days in a sprint are dedicated to coding and the actual development of software. To start off each day, a very short meeting is arranged, during which the developers take turns presenting what they are currently working on and what they plan to do (the rest of the day), as well as discuss any difficulties encountered. The purpose is to catch any impediments to the development as quickly as possible. This meeting is referred to as *daily stand-up meeting*, and is commonly arranged in standing position in order to keep it short and concise.

At the end of the sprint, there is a meeting, known as a *retrospective meeting*, where the team gathers experiences, both good and poor, from the current sprint. The team can easily assess which activities or practices worked well and which practices/activities need to be improved, as well as discuss possible solutions to the most critical problems. Another meeting, the *sprint review*, is also arranged near the end of a sprint/iteration. There, the tasks implemented in the current sprint are presented to interested customers of software.

For a more thorough description of specific Scrum practices, see section 4.5.1 and table 4 in section 4.5.

1.4.4 Extreme Programming

Extreme programming, or just XP, is another well-known agile methodology. Analogous to Scrum, it focuses on communication, close relationships between customers and developers and short-time iterations. XP describes in great detail the preferred work practices. Among the most important are pair programming, frequent code review and testing, both unit testing and user acceptance testing.

The organizational aspects found in Scrum are mostly absent in XP, as the methodology is more focused on work methods than on the general organization of a project. The methodology does not prescribe any roles, but does uphold that the team members should view themselves as equal peers of developers, in order for all to be able to implement any of the requirements. XP has been criticized for its inability to scale up to large projects, and has not been proven successful (at least not to the same extent as Scrum) in projects with many participants [11].

For a more thorough description of specific XP practices, see section 4.5.1 and table 4 in section 4.5.

1.5 Thesis outline

As well as providing certain definitions, the first chapter introduces the main research questions of the thesis and the motivations for investigating these. In chapter 2, prior research into the development of scientific software development is presented. Chapter 3 describes the research methodologies, the systematic review and the case study. Chapter 4 describes different approaches in terms of measuring agility in the projects and the chosen technique (later utilized in the systematic literature review and in the case study), as well as the rationale for choosing this particular approach. Chapter 5 contains the systematic literature review on agile practices and their effects in scientific software development. Next, the case study is presented; chapter 6 describes the case study, its cases and its purpose, while chapter 7 presents the results of the study. In chapter 8, the results of the literature review and the case study is discussed. The next chapter contains a presentation of the limitations of the study (including validity threats). The last chapter, number 10, summarizes the results and presents conclusions to the research questions.

2 Scientific software development

The purpose of this chapter is to present a summary of already existing research on scientific software development, as well as identify common challenges in such development. There are some studies aiming to characterize scientific software development on a general level. Such studies, primarily multiple-case studies or surveys, have been emphasized in this chapter. A few studies focusing on delineated aspects of scientific software development are also included.

2.1 Research on scientific software development

Software engineering research has traditionally focused on techniques, methods and concepts that are applicable more or less independent of the context and local conditions of a software development project. Scientific software operates in very specialized domains, limiting the usability of general development models or at least complicating how such models may be customized to fit the settings and requirements of a particular project. Additionally, scientists are rarely fully engaged in software development (especially when the project is non-commercial) and the teams are often ad-hoc, e.g. scientists only write software if they themselves need the new/improved functionality.

Researchers, both within the software engineering and the scientific software communities, have recently started investigating the nature of scientific software development. A generalization of scientific software development is perhaps impossible to obtain, as the processes therein are diverse and only to a small extent formalized. Nevertheless, some general aspects and factors have been identified.

Diane Kelly describes the perceived gap between software engineering and scientific software development in [1]. She points out that software engineering originates from scientific computing. Since the early days of computing, there has been an increased focus on general methodologies, concepts and domain-independent practices, often in non-scientific settings. According to Kelly, many of these general aspects are simply not relevant for scientists developing software. Nevertheless, scientists “need appropriate software engineering knowledge to support their work” [1], meaning that the described “chasm” must be bridged for scientific software development projects to successfully adopt ideas and concepts from software engineering.

2.2 Surveys on scientific software development

A recent survey [8] by Hannay et.al. provides statistical data on the development of scientific software. Almost 2000 responded to the online survey, making it one of few studies revealing general trends in development of scientific software. The authors discuss and analyze the data, providing us with important insight into the common practices found in the development of

scientific software, as well as increased understanding of scientists' perception of important software engineering concepts.

Nearly all the survey participants were of the opinion that developing software is an important facet in their research. On average, about a third of their work time is dedicated to such development, an amount of time which has increased in recent years. In terms of learning software engineering concepts and acquiring programming and development skills, the scientists regard self-study and peer learning as far more important than formal education and training. As to user community scale, the authors found the statistical data to support their hypothesis which stated that the number of users of the software is either very small (less than 3 users) or very large (more than 5000 users).

The survey results in [8] show that the understanding of, and the routines/practices related to a software engineering concept does not necessarily match the perceived importance of that very concept. The biggest mismatch in that respect occurred for testing concepts, with as much as 13.6 percentage points difference between those having good command of testing-related activities and those that regarded this as important. A difference was observed in other concepts as well, such as software verification and software construction.

There is a possible correlation between size of the development project or team and the ranking of importance of central software engineering concepts. A challenge is found in defining and specifying the requirements, and the appreciation of this problem tends to be greater when developers use a considerable amount of time developing software or is a part of a large development team. In smaller teams and projects, aspects like software requirements and project management are less frequently identified as challenges.

Development methodologies in scientific software are the topic in a survey conducted by Arno F. Granados. Two articles have been written about this survey. The first [12] describes the survey design and the motivations for carrying out the survey. The other [4] presents the results. Unfortunately, there are no discussions or analyses related to the reported results; only the percentage scores of each question is presented, which in turn makes it impossible to identify any correlations between the different questions. There were 60 respondents to the survey, all of them attendees of Astronomical Data Analysis Software and Systems (ADASS) in 1998. Thus, the study is primarily based on how astronomers develop software and is therefore not as general, neither in sample size nor domain, as the previously mentioned survey [8].

The study [4] shows that an incremental or an iterative development approach is most common. The respondents viewed the effectiveness of project management and the usefulness of the applied development model favorably. Both these aspects were regarded satisfactory by 77 percent of the participants. The development teams are usually small-sized (less than seven developers), but occasionally have more than 10 developers. Developers are often highly educated; as much as three fourths of the respondents hold a master's degree or higher.

It is reported that when projects were unsuccessful, it was, in half the cases, due to problems with requirements; either insufficient specification of these in the initial stages, or changes in these during the course of the project. Another interesting finding was that most of the respondents were involved in several other software projects simultaneously. In fact, only 32% of the respondents participated in just a single project. Testing (unit, component or system test), requirements and code review activities were also addressed in the survey. None of these activities were much used. The respective activities were performed by 43% (testing and requirements activities) and 20 % (code review) of the respondents.

The participants in both the first [8] and second [12; 4] survey are highly educated scientists who actively develop and use scientific software in their daily work. The surveys differ somewhat in terms of how the questionnaires are designed and how the results are presented and analyzed. The results from the studies seem to match very well and there are some clear common trends pointed out in both studies. No major deviations between the studies were identified. Requirements and associated activities are identified as challenging, and may be significant adversaries to the completion and/or success of a project [4].Testing activities are not necessarily conducted systematically (in some projects not even at all) [4] and are identified as challenging. Testing is in fact the activity displaying the greatest mismatch between perceived importance and appreciation by the scientists [8]. Some aspects are only addressed in one of the studies, such as code review, scale of development teams and number of users, making it harder to identify common trends for these.

2.3 Case studies on scientific software development

Carver et. al. presents in [2] a characterization of scientific software development models, based on a series of case studies. A total of five projects (cases) were examined by the researchers, in order to investigate shared commonalities. Other goals were to determine activities or practices contributing to success and to identify, as a contrast, activities and practices that were being counterproductive.

Great variations occur in terms of team size and user community size. The number of full-time employees working in a software project is between three and fifteen, and the number of customers varies as much as from zero in one project to several thousands in another. Despite a few differences between the projects, there are some common features and attributes. All of the projects have large code bases, with more than 100 000 lines of code. Fortran and C++ are the most popular programming languages. The choice of programming language does not change over time; projects that have lasted for ten or more years primarily use Fortran, while newer projects tend to use C++.

The study [2] identifies aspects of the development which were perceived difficult to handle properly by the scientists. Verification and validation were widely recognized challenges. As the authors point out, these activities may be impossible due to the nature of scientific projects, where the desired result may be unknown before development starts. Due to this

uncertainty, determining the requirements (of the software) also represent a challenge. Requirements need to be elicited and defined along the way.

One of the main differences between scientific software development and regular software development is the complexity of the software. This means that the development teams must basically consist of scientists having a profound understanding of the specific scientific domain. As the gap of knowledge between the scientists and the computer scientists is vast, there are relatively few of the latter kind participating in the projects. The teams often found it easier "for the domain scientists to write software than for the software engineers to learn all of the relevant science" [2]. Some of the teams are multidisciplinary and consist of both scientists from the specific domain and computer scientists, but the latter group constitutes less than 20% of the projects' members.

Requirements are discovered during the course of the project – and to lesser degree understood or specified in the beginning. Although no explicit process model was used in any of the projects, the processes tended to be somewhat agile-oriented. This may be due to the challenges associated with stipulating requirements. Cultural reasons may also play a part in this, as the scientists in the study "tend to view 'process' unfavorably" [2] – "process" in this case meaning *prescriptive process*.

A report on the development and use of scientific software has been conducted by Rebecca Sanders in her master's thesis [3]. The basis for the analysis is a selection of software projects from a wide array of domains. Due to variations in the projects' domains and practices, there was no uniform manner in which to describe the development of the projects investigated. Nevertheless, by examining the specifics of each project and performing a comparative analysis, Sanders was able to map out certain key characteristics. Based on the acquired characterization, she is convinced "that scientists and software engineers approach software development in different ways".

The evidence in the thesis [3] was gathered from interviews, arranged to collect personal experiences from the scientists in the participating projects. A total of sixteen persons were interviewed; thirteen developers and three users. The interviews were open-ended with no specific set of questions, yet conducted in such a fashion that certain key aspects of development (among them, "purpose of software", "requirements documentation" and "testing") were always covered. The interviewees were encouraged to put special emphasis on the aspects of development most relevant to their project. All of the interviewees were highly educated; a majority being professors in their respective scientific domains.

All the projects had an iterative development process to accommodate either changing theory or changing scope of the implemented theory. The life-span of the processes was either very short or very long, ranging from a few weeks to several years (or even decades). Although some projects seem to be of considerable size and have a long life-span, there was seldom any particular focus on design of the software. Some developers referred to their software as a "behemoth", meaning that the code was a culmination of the effort put in by many scientists over a long period of time, making it very hard to fully understand and maintain the software.

Some challenges were repeatedly mentioned by the scientists during the interview sessions. The most prominent ones were lack of code design and code review, as well as inadequate handling of requirements and testing.

None of the projects had formally established a means of dealing with code design, leaving the software hard to maintain and enhance. Only four of the interviewees performed any kind of code review. Requirement specification and handling also emerged as a difficult aspect, with many drivers for change; scientific theory, the scope of that theory or in the implementation. In many cases, the scientific theory itself represented the requirements. If any formal specification of requirements were indeed necessary, it was almost exclusively created in the final stages of the development, near completion of the project. Testing activities were heavily emphasized by the interviewees. Although extensive testing in some cases may not be of great concern, there was a general lack of commitment to such activities. Validation testing and usability testing were the only forms of testing frequently used by the developers. The complexity of the theory behind the scientific software may impede testing, as it is difficult to create reliable *oracles* (i.e. difficult to ascertain what the correct output should be).

Judith Segal has performed a series of field studies on the development of scientific software in recent years, the general trends from which are summarized in [13]. The purpose of the studies was to identify software development models practiced by the scientists and other members of the project teams. The domains investigated were financial mathematics, earth and planetary science and structural biology, elaborately discussed in separate studies – more specifically in [14], [15] and [16] respectively. Segal goes on to discuss how the models deviate from models found in traditional software development. Another research objective was to investigate how well scientists collaborate with software engineers, and identify challenges occurring when software engineers have joined the development projects.

There are basically two contexts encountered in the field studies. The most common is development intended for either personal or close colleagues' use. Alternatively, the software might be intended for the larger scientific community of which the developers are an integral part. In both contexts the development is performed by a closely connected group of scientists/developers. During the course of the project, the science unfolds and the understanding of the software evolves. The development processes were characterized as iterative or incremental, where requirements (at least a significant part of them) are being discovered as the development proceeds. The scientists usually have very strong intuition as to the purpose of the software and they rarely gathered/defined any detailed requirements or tasks. The same applies to software evaluation. Neither of these activities were conducted in any systematic manner.

Handling of requirements is not a big problem in teams exclusively consisting of scientists, due to their shared strong intuition of what the software should be and their understanding of the scientific domain. It is when software engineers have been part of the projects that problems have arisen in that respect. The software engineers do not share the expertise of the science and have other demands regarding collection, elicitation and specification of

requirements. The scientists' informal manner of resolving such issues does not match these demands. There were other evident challenges as well, in the projects where scientists and software engineers collaborated closely; scheduling of the project and testing, portability and maintainability issues.

The case and field studies report that scientific software development projects are often of considerable size and that the life-cycle of such projects may be very long. The long life-cycles may explain why Fortran is extensively used, as projects that have been around for a while tend to use Fortran, whereas newer projects often use C or C++. It is hard to identify a set of activities, transitions between activities and roles in the projects, as the development processes, generally, are far from formalized. The authors of these studies think that the actual processes do not bear much resemblance to plan-driven development approaches. In fact, the applied practices are more similar to those prescribed by agile methodologies. This is perhaps mostly due to the requirements, which are discovered during the course of the project [2; 13]. Although the projects need to take this into account it is not uncomplicated to customize the projects to accommodate requirements changes. This aspect of the development can be very challenging indeed, and is considered one of the top problems in scientific software development by all the studies. Another challenging aspect, mentioned by nearly all the articles, is testing. It is very difficult to establish proper testing routines. Several other aspects are also identified as difficult in one or more of the articles, such as code review [3] and scheduling, portability and maintenance of the software [13].

These case and field studies provide a clearer overview of how scientific software is developed and the general impressions and results are agreed-upon and mutual. The studies do not display any particular deviations with regards to the results from the abovementioned surveys. In fact, the case/field studies seem to approve many of the trends indicated by both surveys, with regards to the applied processes and the common challenges facing scientific software projects. Certain aspects not addressed in the surveys, are covered in greater detail in the case/field studies.

2.4 Studies on specific aspects of scientific software development

The interviews from Sanders' thesis were also used in another article [17], the focus of which is risks associated with scientific software. The study indicates that the development of scientific software faces massive challenges (or risks), due to highly complex scientific domains. There are challenges relating to both the underlying theory of the software, as well as the implementation of that theory. In addition, the mere use of the software poses as yet another area of risk which must be taken into consideration. The authors discuss how these risk areas affect the applied development process. In that respect, they consider design, documentation and testing practices to be influenced by the risk factors.

There is a broad agreement that testing of the software is important, but the practices and routines related to testing are, in general, not consistent or well-organized. Both the implemented theory and the implementation are subject to review and testing. The motivation for writing the software influence testing decisions; in many cases the purpose of the software is to prove (or test) a scientific theory. The scientists tend to focus on the theory (which can be described as “unwritten requirements”) rather than on the code or implementation, when there is a mismatch between the output and oracle (i.e. expected output).

The design, in particular code design or architecture, may turn out to be a problem when the programs become large or many people are involved in the development. An iterative approach, where modules are added to a possibly “behemoth” program, is common. In such cases, the scientists do not consider redesigning or refactoring of the code as a valuable use of their time, as they are more interesting in doing science. The documentation produced in the projects is related to the scientific theory rather than the software. In cases where the theory is very established, and the program utilizes scientific theories rather than examining them, the focus of documentation is shifted towards the code.

An article [18] by Decyk gives us important insight into why the Fortran programming language is widely used in scientific communities. The programming language seems to correspond well with the procedural nature of the science they are implementing (for instance complex multi-step algorithms). How well the study reflects the current situation in scientific software development is uncertain, as most recent projects opt for other languages (such as C/C++ or Java) as their primary programming language [2]. Nevertheless, the use of Fortran will probably prevail for some time as projects tend to have long life-cycles and rarely replace their programming language [2].

2.5 Synthesis of general trends and challenges

Scientific software operates in very specialized domains. Diane Kelly suggested that the domain-specificity of science might explain why results of research in SE have only rarely been oriented toward scientific computing [1].

Scientists use their software to do complex calculations or simulations, if not to test and explore scientific theories. These characteristics of scientific software entail that, in contrast to the development of, say, administrative or business enterprise software, the writers of scientific software cannot determine what the correct output of an application should be in the traditional sense. Also, the software may evolve through the combined effort of a number of scientists over the course of many years, continuously adding new functionality to the system [17]. This poses particular challenges from the software engineering point of view: First, requirements elicitation and specification must be highly dynamic. Due to the exploratory nature of many scientific projects, the elicitation and specification of requirements is problematic because they may be unclear, or even unknown, up-front. Secondly, since

requirements are of such a volatile nature, one should expect that testing the software with regards to such requirements would have to adapt to changing requirements.

Thus, *a priori*, inherent characteristics of scientific software would seem to impede requirements handling and testing in the outset. In fact, the lack of knowledge about requirements and testing principles has been identified as problem areas in several studies [2; 3; 8]. In the first survey, it is apparent that requirements activities are perceived as problematic in scientific software projects, especially when the teams are large or when scientists dedicate much time to developing software [8]. The definition of test cases for validation and verification of the software is perceived as challenging. For example, it is often not obvious to stipulate whether an error relates to the scientific theory or to the implementation (numerical approximation) of that very theory. Among many of the participants in the survey [8], testing-related activities were indeed regarded as an important part of the project. However, there was a considerable difference between the number of survey participants having said opinion and the number of survey participants having good command of such activities. Consequently, testing skills seems to be a weak point for scientists developing software.

In most aspects of the development of scientific software, the urge to conduct science is the primary motivation and goal. Scientists seem to have a different approach to developing software than software engineers. The development method used is usually one that has emerged as best practices based on the team members' experience [2]. Also, the variation in domains and motivations found in scientific software projects are factors that influence the development. Consequently, one would expect large variations in development methods both across and within the different scientific domains.

Nevertheless, some common ground may be found, and due to the challenges in determining requirements up-front and the subsequent testing, scientific software development may lend itself more easily to agile-oriented practices than plan-driven practices. Sanders supports this notion by stating that most projects under investigation in her study had an iterative, rather than a plan-oriented, approach to development [3]. However, she does not state explicitly which specific practices that are applied.

3 Research Methodology

Two research methodologies are used for addressing the research questions; a systematic review and a case study. Both these research methodologies are well suited to explore compound research questions of the type proposed in this master thesis. In this chapter the two methodologies, as well as the rationale behind the execution of these, are presented. Prerequisites needed for carrying out the studies are also covered in this chapter, and the reasons for choosing these two specific methodologies are explained.

3.1 Prerequisites for conducting the studies

In order investigate the research questions, a method for assessing and evaluating development projects from an agile viewpoint must be established. As there is no *de facto* standard for such evaluations, an appraisal of available approaches is necessary, in order to determine an agile measurement technique appropriate for the task at hand. This appraisal is presented in the subsequent chapter, and the obtained agile mapping chart is used in both systematic literature review and case study.

In order to complete the systematic literature review, access to a selection of literature databases has to be granted. The following databases are the primary sources of material: *Association for Computing Machinery (ACM) Digital Library*, *Institute of Electrical and Electronics Engineers (IEEE) Xplore Digital Library*, *ScienceDirect* and *Institute for Scientific Information (ISI) Web of Science.* Access to all of these databases is granted by the University of Oslo. *Google Scholar*, which is a publicly available search engine for research literature, was also used in the literature review.

To conduct the case study, access to a number of scientific software projects had to be arranged. One of my supervisors provided me with the necessary contact information and initiated communications with key members from three different projects, who in turn helped arranging the interviews. Prior to the conduction of the interviews, the agile measurement technique had to be established, in order for the interview guide to be designed in such a way that all necessary agile aspects were covered.

3.2 Systematic review

Systematic review is an important research methodology in evidence-based software engineering and aims to identify and appraise all relevant studies already conducted about a particular topic of interest. The appraisal focuses on both the quality of the studies and the relevance of the studies' reported results with regards to the specific, proposed research question(s).

Both an article [19] by Dybå et.al. and a technical report [20] by B. Kitchenham focus on systematic reviews in software engineering research. The first one [19] is an experience report on applying systematic reviews, and put forth a set of guidelines on how to conduct such reviews. The technical report [20] does likewise, but is perhaps more extensive and general. Both of these sources were very useful and provided detailed, yet not too rigorous, guidelines on how to perform systematic reviews.

In [20], B. Kitchenham reviewed three existing systematic review guidelines, primarily used by medical researchers. This was done in order to ascertain a set of guidelines for systematic reviews adapted to the realm of software engineering research. In these guidelines, the effort of performing a systematic review is divided into three main stages: *Planning the review*, *Conducting the review* and *Reporting the review*. The first stage consists of two sub-elements, namely *Identification of the need for a review* and *Development of a review protocol*. Reporting the review, the third stage, is a single step. The most elaborate stage is the second one, which consists of five separate phases (these stages are used in [19] as well), as seen in table 1.

1	Identification of research
2	Selection of primary studies
3	Study quality assessment
4	Data extraction and monitoring
5	Data synthesis

Table 1: Stages of conducting a systematic review

3.2.1 Identification of research

Designing a search strategy that (hopefully) identifies all relevant studies on the subject is the first stage of conducting a systematic review. Included in this stage is to establish a search query (or a set of search queries) and determine where (which literature databases) to execute this search query. The search query in the review was designed by assessing some initial trial searchers, as well as inspired by the example queries presented in [19] which focused on a semi-similar research topic. The sources of evidence used are the literature databases mentioned in section 3.1.

The ultimate aim is to find all relevant studies when putting the search queries through the different databases, while at the same time limit the number of irrelevant studies in the results. Articles concerning use of agile methodologies do not necessarily have a mutual, consistent terminology. In addition, many of the instances of agile methodologies (i.e. specific methods such as Scrum, Crystal and XP) have names which undeniably are more likely to occur in studies of no interest to the research questions (this was particularly the case for some of the methodologies). In cases of astounding numbers of search results, refinements provided by the different digital libraries' search engines were used. The search is documented thoroughly; the number of results and the number of relevant/irrelevant studies obtained from each sub-query from each included research database, and the summary of results for the complete query from all databases, are reported in chapter 5.

3.2.2 Study selection

The stage of study selection focuses on the criteria for inclusion or exclusion of a single study in the review. The purpose of this step is to obtain a final list of review elements, which are to be subjects in the later quality assessment. The articles included in the review have to report results relevant to the research questions. To determine whether a study actually does this, it must go through some stages of analysis. First, the titles will be investigated, in order to exclude clearly irrelevant ones. Next, the abstracts of the remaining studies will be examined, constituting a more thorough assessment of the studies' relevance for the research question. Full copies of the still included studies have to be obtained at this point, and each of them will be examined through thoroughly in order to evaluate the content in its entirety. If the study still proves to be relevant to the research question and topic of interest at this point, it is included in the final set of review objects.

This stage of the review is also documented along with the statistics from the search queries. How many articles excluded based on title, abstract and overall content from each subquery and database are reported and presented in chapter 5.

3.2.3 Quality assessment

The quality of the relevant studies needs to be assessed. This is done primarily in order to investigate whether the results presented in the studies are reliable and applicable to the proposed research question(s). Another motivation for examining the quality of the studies is to identify which studies to emphasize when summarizing the overall results in the synthesis. The quality assessment may be divided into three separate groups [20]: *Bias, Internal Validity* and *External Validity*.

Bias is a systematic error in the study, which may affect a study's results, analysis and conclusion in numerous ways if not accounted for. A study is internally valid if it does not have any significant systematic errors or biases. The external validity is ”the extent to which the effects observed in the study are applicable outside of the study” [20]; in other words the generalizability potential of the study. There is a hierarchy in the quality assessment: Absence of bias is a prerequisite for internal validity, and external validity is dependent on internal validity.

In terms of the research question related to the effects of using agile methods, special emphasis have to be put on the cause/effect relationships in the study and how it is reported. The quality of the handling of testing and requirements in a project may be affected by the agile practices actually applied or not, but several other factors may also contribute to eventual good or bad practices in those departments. Whether the potential impact of such confounding variables is taken into account in the studies will be investigated thoroughly.

3.2.4 Data extraction

The first phase of the data extraction process in my review was to record each study's title, publication year and name of journal and authors. Then the results of the studies were recorded, along with its key pieces of evidence. As the research objects in the studies were one or more scientific software projects, relevant information connected to each project was collected, such as name of project, domain of project and development model used (if any). One of the reasons for collecting detailed data about each project, apart from obtaining an elaborate reporting of the findings, is that the same projects (evidence) may be the object of research in several studies (perhaps with somewhat different focus). These data will be collected not necessarily to prevent inclusion of multiple studies using the same evidence in the review, but rather to ensure that these studies do not influence the overall data synthesis or conclusion disproportionally.

The projects from the studies are assessed using an agile mapping chart (presented in the next chapter), which focuses on individual, specific agile practices and thereby evaluate whether these are present or not in the projects described in the respective studies. Hence, data and evidence about specific agile practices in the project will be collected. The descriptions of how testing and requirements activities were done in the projects, and whether the practices related to handling of such activities were regarded successful or not, are also gathered. Also, in cases where the effects of using agile development models or practices are discussed by the authors, data were gathered about other factors potentially affecting testing and requirements activities in the projects.

3.2.5 Data synthesis

In this stage of the review, the results from the individual studies are compared and analyzed. The different pieces of evidence are discussed, as well as the studies' combined ability to answer, or shed light on, the research questions proposed by the systematic review. In some ways, this stage is a conclusive step where all the studies' results, relevancy, validity issues and overall quality will be accounted for and considered in order to answer the proposed research questions. If many studies are still included at this point, quantitative measurements may be used. Such quantitative measures were not used in my review, as the list of relevant studies only contained five elements.

The articles are then compared in terms of the reported results and the relevance and reliability of these results. The ultimate purpose of the review is to obtain answers to a number of research questions or hypotheses. How general and reliable such answers are depends on the sheer number of relevant studies as well as on how much conformance exists between the different review objects. Eventual deviations from the trends are examined in order to analyze why these studies reported unusual results. This is possible due to recordings made during the data extraction stage, where such evidence was gathered. The final

assessment relates to whether the results from the studies are capable of answering the research question(s) sufficiently and in how general and confirmatory a manner.

3.3 Case study

As described by Yin, a case study is an in-depth examination of a selection of one or more contemporary phenomena within a real-life context [21]. In my thesis, the case study focus on the processes applied in a few scientific software projects, with emphasis on eventual agile elements. Like other research methods, a case study may be exploratory, descriptive or explanatory. The case study in this thesis is primarily exploratory when investigating the software development processes, and descriptive when mapping and evaluating these processes from an agile viewpoint, according to the agile mapping chart. The systematic review is of a more explanatory type, as it investigates the effects (on testing and requirements activities) of using agile practices. In the outset, the case study does not share this focus, but any such effects will be investigated if a project incorporates one or more of the related agile elements/practices.

In this thesis a holistic multiple case-study [21] is conducted; three different cases are investigated within their specific contexts. The primary reasons for choosing the three specific projects were access (via one of my supervisors) and availability in terms of geographical localization of the project and interviewees. There are both advantages and disadvantages with conducting a multiple case study rather than a single case study, but it may be preferable if one has the choice [21]. The primary advantage is the possibility of replication, meaning that “analytic conclusions independently arising” from more than one case indeed is “more powerful than those coming from a single case” alone. [21]

Depending on the context, a number of evidence sources may be relevant in the data collection phase of a case study. Yin emphasizes six different types: *Documentation*, *Archival records*, *Interviews*, *Direct observations*, *Participant-observation* and *Physical artifacts.* In my case study, the data collected were interviews. The primary reason for this is that other types of evidence sources would probably not have been consistently accessible in all three projects. For many of the abovementioned evidence types it is also questionable that these would yield any information relevant to the research questions. Interviews focus on the particular research questions and are regarded as an insightful source of evidence, providing perceived causal inferences and explanations [21], which is surely needed when examining a somewhat abstract concept such as a development process, of which the interviewees (as they are scientists and not professional software developers) may have a limited understanding. This is also the case for the specific agile practices in the agile mapping chart.

3.4 Choice of research methodologies

The two research methodologies were chosen due to their suitability for the type of research questions proposed by this master thesis. Unfortunately, it is hard to conduct a sufficiently comprehensive analysis of both research questions by using only one of the research methodologies. Thus, two different research approaches were used in the thesis, as they complement each other very well, and the probability of answering both research questions increased by conducting both a literature review and a case study. By the choice of research methodologies, both previous research on the subject and the cases available could be utilized to full extent.

Although both studies will address both research questions, there are some aspects which are likely to be hard to determine in the literature review and the case study respectively. For instance, the articles obtained in the literature review might not report very explicitly on which practices are being used in the projects. The difficulties of determining which exact agile practices were employed in the projects from the systematic literature review could be addressed more directly in the case study. By conducting semi-structured in-depth interviews, each of the practices could be evaluated and discussed with several participants in the projects. However, it is doubtful that every project in the case study is well-suited for assessing the effects of agile practices (research question 2), as it is not certain that these projects incorporate agile practices related to testing and requirements.

A series of different research methodologies have to be assessed prior to conducting a study of any kind, in order to find the methodology, or methodologies, most appropriate for the research questions and available research objects (in this case: the three scientific software projects). The systematic review was chosen for its in-depth focus on a complex phenomenon and also because the desired type of evidence (scientific software projects using agile approaches to development) is not easily accessible. This means that the analysis needs to be founded on prior research on the matter.

To investigate FEniCS, Dalton and Olga, where participants were available for interview sessions, there were essentially three approaches to consider; an experiment, a history or a case study [21]. As suggested by the name, a history is more focused on historical events as opposed to an experiment or a case study, which emphasize contemporary events or phenomena. In order to conduct an experiment, the investigator has to be able to manipulate the behavior of the objects under examination (in this study: the processes of scientific software projects). As the processes in the projects in the case study could not be manipulated in any way, just examined to understand the software development processes and moreover the agility present in them, the case study turned out to be an apt research methodology.

4 Measuring agility

This section covers an appraisal of different, available agile measurement techniques, methods and frameworks. In order to find a proper measurement technique to use in the remainder of this thesis, agile evaluation frameworks in literature are explored. Some online approaches, not described in academic literature, are considered as well. There also exist some similar studies, where evaluation of agility is central to the research; techniques in these studies are also assessed. All approaches are presented briefly, before their applicability is evaluated. The final solution in the end was to create an agile mapping chart, based on Scrum and XP practices. The agile mapping chart is presented after the other approaches are assessed, along with the rationale for choosing this approach over other alternatives.

4.1 Introduction

A number of software engineering concepts are quite difficult to measure. This applies perhaps especially to software development processes, which are complex and somewhat abstract. The actual applied processes, or descriptive processes, in a software project are seldom or never in total conformity with the chosen methodology (or prescriptive software development process), if any such is selected at all. Hence, it may be challenging to map a descriptive process to its corresponding methodology.

In the aftermath of the introduction of agile methods to the software engineering scene, a number of evaluation frameworks for development methodologies have emerged. As the sheer number of agile methods, as well as the popularity of these, increased, more extensive frameworks was needed to assure that all agile-related aspects were accounted for. This was also needed in order to compare agile process models to other development models.

The measurement techniques found in literature are primarily intended for determining which process model family a methodology corresponds to. Some articles contain detailed, procedural techniques explaining how to conduct such assessments. Even though these measurement techniques do not automatically transfer to investigating a single, descriptive process, they may still be relevant background material. At the very least, the key characteristics used for assessing the agility may be elicited, and the tools being used in the evaluation are also of interest.

4.2 Agile Evaluation Frameworks and Techniques

Taromirad and Ramsin define in [22] a set of meta-criteria in which a process evaluation framework should fulfill. They used these in an appraisal of existing frameworks. The set consists of meta-criteria of three general types:

1. *"Features that any evaluation criterion set should have."*

2. *"Features that an evaluation criterion set needs for evaluating a software development methodology."*
3. *"Characteristics needed for evaluating agile-specific features in software development methodologies."*

Agility features, such as speed, flexibility and frequent releases, are all taken into consideration, and is addressed by meta-criteria of type three. None of the appraised evaluation frameworks in the article did meet all the proposed criteria, and some were in fact "lacking in several aspects" [22]. There were short-comings in the agility-related evaluation criteria; such criteria were not addressed properly. The conclusion of the appraisal was that new evaluation frameworks, matching the defined meta-criteria, were highly warranted.

4.2.1 CEFAM: Comprehensive Evaluation Framework for Agile Methodologies

The same authors, Taromirad and Ramsin, took matters in their own hands and created CEFAM [23], an extensive evaluation framework. This framework covers all meta-criteria defined in [22]. CEFAM is an abbreviation for "Comprehensive Evaluation Framework for Agile Methodologies", a name which clearly declares their ambition of addressing the abovementioned shortcomings of existing frameworks. The agility evaluation criteria are most relevant (for the purpose of this thesis), as the article claims that these "can be used to evaluate the degree of agility in any software development" [23]. Aspects like speed of producing results, sustainability of that development pace, flexibility and responsiveness are all addressed. Included in the agile-related criteria is also learning, or self-improvement in the project work, as well as user collaboration. The framework may be an asset when assessing individual processes rather than methodologies, and used in such contexts as well.

4.2.2 Goal-based agility assessment

In [24] a different approach to agile assessment is presented. The article does not describe a framework considering the meta-criteria in [22]; it is rather a more practical assessment tool directed towards assessing a single process. Rather than imposing measurements based on a set of criteria, the authors propose the use of project goals as the metrics in the agility assessment, as these may be customized according to the unique and specific conditions of a project. As these goals are business or context dependant, there is no set of goals provided, just illustrative, example-goals like "Share knowledge" or "Quick Releases", meaning that goals have to be tailored to the project in question.

4.2.3 Agility Measurement Index(AMI)

Another article [25] describes a technique for obtaining a numerical measurement unit of agility in a software project. The unit is intended for determining which process methodologies that is suitable for the project in question. Agility Measurement Index (AMI) focuses on five dimensions of a software project: duration, risk, novelty, effort and interaction. The rating of the dimensions is up to the user of the technique; each dimension is assigned a minimum value, usually one, and a maximum value (which the evaluator chooses himself/herself). Using the interval between the minimum and maximum value as possible scores, the evaluator assigns a value to each dimension. The unit of measurement is then calculated by dividing the sum of actual scores on the sum of maximum values, determining a percentage-based score of how agile the project is.

4.2.4 The Karlskrona test

The Karlskrona test [26] consists of a number of simple multiple-choice questions with 4 alternatives. There are eleven questions in the test, which focus on different aspects of the development process. Each individual answer/alternative to a question has a value of either zero or one, depending on how agile the alternative is (agile alternatives are valued to “one”, while non-agile alternatives are valued to “zero”). Thus, a total test score between zero and eleven is obtained when applying the technique. A score of zero points indicate that no agile elements at all are present, whereas a test score of eleven points indicate that the process may be characterized as agile in all aspects addressed by the test. A five-point scale is used to categorize how agile the team is – depending on the total score. The individual answers may in turn be used to identify separate process elements of agile (or non-agile) nature, as two of the alternatives to each question indicate an agile approach, and the two remaining denote that the element is more similar to how it is handled in a traditional, waterfall-type development model.

4.2.5 The 42-point test

The 42 point test [27] is another agile evaluation method. The test is quite similar to the abovementioned Karlskrona test, only that is consist of 42 statements. The user of the method should assign each statement to either true or false. Each statement classified as true gives one point. This test does not provide a scale, for interpreting the score, like the Karlskrona test does. The set of questions is fairly large compared to the Karlskrona test, and the test constitutes a more fine-grained separation of process elements - and characterizes each of them as agile or non-agile.

4.2.6 The Nokia test

Another alternative is the NOKIA test [28], originally created by Bas Vode and later refined by Jeff Sutherland (one of the creators of Scrum), which is a two-part pass/fail test of whether the project in question is agile or not. The first part tries to answer whether the process in question is iterative. This evaluation is based on three conditions; the length of the iterations, the specification of the iteration's tasks and last the state of the functionality at the end of iteration. Part two investigates whether the project performs the specific agile development model Scrum with:

- Everybody knows who the Product Owner is.
- Prioritized product backlog (list of defined tasks)
- The team have defined estimates on the tasks in the product backlog
- The team keep track of velocity and create burn down charts
- Nobody (not project managers or others) impedes the team's ability to work

Like the 42 point test, the Nokia test is very focused on Scrum. Arguably, it is possible to achieve a high degree of agility even though the team does not adhere explicitly to every Scrum practice. For instance, an absence of burndown charts would mean that the team failed the NOKIA test, even if they pass all remaining criteria. Another problem, which may apply to some of the other tests as well, is that it considers certain aspects as either agile or not, when the actual situation is probably much more complex – if aspects is handled in a semi-agile approach it is unclear what decisions one should make.

4.2.7 Adoption of XP practices in the industry - a survey

The specific practices proposed by XP are not generally applicable. Some practices are directly suitable to most projects, whilst other practices may need tailoring to fit the project or be discarded completely as they are too hard to adapt. These premises and notions are investigated in a study performed by Bowers et.al., in which they evaluate the adoption of XP practices in a series of software development projects [29]. Fourteen representative case studies form the basis of the study. Each project is analyzed and the adoption of each separate XP practices is evaluated. The practices are evaluated to either *yes* or *no*, depending on whether they are present or not, but the authors also examine and discuss how the practices are employed. The projects are compared at the end, obtaining an overview of how XP is adopted in the software industry.

The authors found that most XP practices were "variably applied" in the projects investigated. This is the case for certain aspects central to agile development methodologies, such as on-site customer and user acceptance testing. Some practices are very rare indeed. Among the least used is the system metaphor, perhaps due to its abstract nature and the difficulty of recognizing and formalizing an overall metaphor for the project. On the other hand, some practices also stood out as very popular, especially unit testing and test-driven development, which were used in nearly all the projects investigated.

4.2.8 The RDP technique

There are essentially two ways of defining XP; either by its practices (as in the previous article [29] and in this thesis) or by its underlying rules. Defining XP by its rules has been done by Mirakholi et.al. in [30]. The authors provide the definition, along with a proposed technique aiming to customize XP to the specific conditions of a project.

The rules of XP are classified into two categories: ***Rules of engagement*** and ***Rules of play***. The five rules of play are based on aspects/values related to XP (such as *Communication, Simplicity, Feedback,* and *Courage*), while the six rules of engagement are based on what "make you agile", meaning that they are grounded on the fundaments of agile software development rather than a specific agile methodology (such as XP, Scrum etc.). The rules of play and rules of engagement are presented in table 2 and 3 respectively. The authors found it problematic to define XP by its practices due to a couple of reasons: Firstly, certain practices may be unwanted or unnecessary in the project. Secondly, some practices have to be adapted or customized to fit the development environment.

Rule#	Rule of play
P1	Continuous testing
P2	Clearness and quality of code
P3	Common vocabulary
P4	Everyone has authority and at least two people have understanding to do any task
P5	Test-first programming in pairs

Table 2: XP rules of play

The technique proposed in the article is called RDP (Rule-Description-Practices), which is a mapping between XP rules and XP practices. The fundament of the technique is the previously defined rules, which are being analyzed one by one. The users of the technique identify and select practices matching or satisfying each rule. Each practice, both original XP practices and self-selected ones, may be used and they can cover multiple rules. Each rule may also be mapped to several practices. After the steps of the RDP-procedure is carried out, a list of practices is obtained, which constitutes the project's customized XP practices. A case study is presented in the article as well, where a software project applied the technique successfully.

Rule#	Rule of engagement
E1	Business people and developers must work together daily throughout the project
E2	Our highest priority is customer satisfaction
E3	Deliver working software frequently
E4	Working software is primary measure of progress
E5	Global awareness/openness
E6	The team must act as an effective social network

Table 3: XP rules of engagement

4.3 Applicability of techniques and frameworks

The CEFAM framework [23] is basically designed for assessing methodologies rather than process instances (which essentially are the research objects in both the literature review and the case study) and is therefore only transitively relevant. Consequently, the technique has to be subject to massive refactoring in order to match the specific purpose of this thesis, and even then the applicability of it is unknown.

It has been pointed out that existing process methodology evaluation frameworks are insufficient with regards to important, agile-oriented aspects [22]. These frameworks are not directly relevant for a single-process evaluation. As the appraisal explicitly point out there is generally an inadequate handling of agile aspects in existing evaluation frameworks as well. Due to these two facts, further investigations into specific frameworks have not been conducted.

Unfortunately, only two articles [24; 25] were obtained related to assessment of agility in a single process. One of them [24] requires quantifiable measures of process elements, yielding it difficult to apply to a process where such metrics may be irrelevant or not even obtainable. The goal-based approach in [24] is, as CEFAM, hard to use in this thesis. It is required that it is possible for each goal to be quantitatively measured. Demands of such rigidity make the technique difficult to apply. In addition, the aim of this agile evaluation is to ultimately become more agile, not necessarily to evaluate agility. To embellish agile aspects and augment them is certainly not the intention of the agile evaluation in any part of this thesis.AMI [25] is perhaps a closer match for the purpose of this thesis, as it is oriented towards evaluating a single process in order to measure its agility. However, to apply the technique it is not straightforward, due to the demand of quantitative measurements, which may prove to be a major obstacle. Additionally, it is difficult to decide the set of elements/practices forming a given dimension, how to weigh these elements up against each other and ultimately how to rank the dimensions. Nevertheless, it is a candidate technique to consider further.

There exist a few online techniques aiming to measure agility. The reliability of such techniques is probably rather low and which references they have used are in most cases uncertain. To my knowledge none of these techniques have ever been used in prior, related research. All the online alternatives aim to measure the agility on a general level. By investigating the separate components of the tests, it can probably be detected which aspects are agile and which that are not. Some of the techniques are very based on Scrum (which reference is unknown) rather than the general agile software development paradigm. The advantage of the techniques is that they are all very accessible, fairly simple to use and that they focus on discrete practices or aspects of the development processes. However, there are also massive drawbacks with these techniques, such as low reliability, uncertainty of which references are used and how complete the techniques indeed are.

One of the best matches in available literature, to the purpose of the agile evaluation in this thesis, is found in [29] . Whether specific agile practices are indeed used is the topic in the literature review and the case study, as well as said article. (It is also interesting to see that they encountered the same difficulties as I did myself in the evaluation (of studies described in articles); the eventual presence or absence of a practice was in some cases impossible to determine, and how confident one can be in each separate evaluation is varying). The approach may very well suit the purpose of both the literature review and the case study respectively. The only culprit is that the reference model is just XP, leaving managerial components of the software development process largely unaddressed. The approach put forth in [30] is quite similar and have many of the same advantages and drawbacks. However, it seems that the focus of the technique (to customize a project's XP practices) and the extra effort of mapping XP rules to specific practices might complicate the matter of assessing the agility in a process.

4.4 Rationale for choice of agile mapping chart

There are several options available for constructing a technique for evaluating agility. It is essential that the chosen technique covers key agile aspects and that it can be used with as few adjustments as possible. Three main options emerged from the appraisal of existing agile assessment techniques:

- Use the AMI measurement technique and rate each dimension equally.
- Use one or a combination of the online alternatives (with proper adjustments).
- Construct a mapping chart based on agile methodologies (similar to approaches used in [29] and [30]). Define the methodologies either by:
    - Their rules
    - Their practices

In order to retain a decent level of abstraction, there are essentially two options: Either defining the agile methodology by its practices or its rules. As seen above, these approaches have been performed successfully, in [30] and [29] respectively. Either one of the approaches could probably have been used, but the final decision was in favor of definition by practices, as this approach is both more accessible and a closer match to the purpose of this thesis. However, the final agile mapping chart has some adjustments and is not exactly similar to the one used in [29]; the reference used for XP describes more practices and practices from Scrum are also addressed.

By merging Scrum and XP practices the agile mapping chart does not rely too heavily on a single methodology. Also, the chosen methodologies are well-established, they are both accessible and there is a certain consensus with regards to their general content and associated practices. Scrum and XP are complementary in the sense that Scrum focuses on practices for management and organization, while XP focuses more on technical development practices or

practical work methods. The combined set of practices addresses a large number of concerns in general software development, while simultaneously capturing the essence of agility.

Using a practice-based approach gives us the desired kind of information. It is of limited interest to obtain a percentage-based unit of the "overall agility" present in the process, as it is the presence of individual elements that are truly interesting. The practice-based solution provides us with that kind of fine-grained identification of (more or less separate) aspects/elements. There is also some comfort in the knowledge that a similar approach was successfully used in another study [29].

Some aspects might have proven challenging to handle in the event of choosing the alternative rule-based approach. Firstly, the approach has not been applied for assessing the agility in a process and the extent to which the technique transfers to that particular purpose is dubious. It is perhaps more appropriate when used to customize practices, as is done in the above-mentioned study [30]. Secondly, it is not trivial to identify the rules and map the practices found in the development process to its corresponding rule (if any). This challenge might be even greater when several agile methodologies are used as references; for instance, additional rules would have to be established for Scrum.

4.5 Agile mapping chart

The agile mapping chart is presented in table 4 below. The table consists of a total of 35 practices. The first part of the table (the first twelve elements) is based on Scrum [6]. The second part of the table is based on XP [7]; more specifically, practices 13-35 originate from XP.

Table 4: Agile mapping chart

#	**Agile practices**
1	Priorities (Product Backlog) maintained by a dedicated role (Product Owner)
2	Development process and practices facilitated by a dedicated role (Scrum Master)
3	Sprint planning meeting to create Sprint Backlog
4	Planning poker to estimate tasks during Sprint planning
5	Time-boxed sprints producing potentially shippable output
6	Mutual commitment to Sprint Backlog between Product Owner and Team
7	Short daily meeting to resolve current issues
8	Team members volunteer for tasks (self organizing team)
9	Burndown chart to monitor sprint progress
10	Sprint review meeting to present completed work
11	Sprint retrospective to learn from previous sprint
12	Release planning to release product increments
13	User stories are written (*)
14	Give the team a dedicated open work space (*)
15	Set a sustainable pace (*)

16	The Project Velocity is measured (*)
17	Move people around (*)
18	The customer is always available (*)
19	Code written to agreed standards (*)
20	Code the unit test first
21	All production code is pair programmed
22	Only one pair integrates code at a time
23	Integrate often
24	Set up a dedicated integration computer
25	Use collective ownership (*)
26	Simplicity in design (*)
27	Choose a system metaphor
28	Use CRC cards for design sessions
29	Create spike solutions to reduce risk (*)
30	No functionality is added early
31	Refactor whenever and wherever possible
32	All code must have unit tests
33	All code must pass all unit tests before it can be released
34	When a bug is found tests are created
35	Acceptance tests are run often and the score is published

Elements marked with an asterisk in table 4 are XP practices from [7], but are also recommended practices in the Scrum methodology [6].

4.5.1 Description of practices

1. Priorities (Product Backlog) maintained by a dedicated role (Product Owner)
Product Backlog is the artifact where all the project's requirements and tasks are defined and prioritized. This list is maintained by the Product Owner, who represents the voice of the customer in the daily development.

2. Development process and practices facilitated by a dedicated role (Scrum Master)
The Scrum Master is the closest to a traditional "project leader". He is the head of the Scrum team and his responsibility is primarily to facilitate for the team to focus on development and the tasks in the sprint; the Scrum Master handles impediments to the development.

3. Sprint planning meeting to create Sprint Backlog
In the initial stages of an iteration a meeting is arranged, which agenda is to stipulate which tasks/requirements will be addressed in the current sprint/iteration.

4. Planning poker to estimate tasks during Spring planning
Planning poker is a technique aiming to obtain task estimates. Each developer in the team gets a deck of cards (which numbers corresponds to an estimate). Upon estimating a task, every developer, at the same time, presents the card they think is correct for the given task. The

plenary estimate is then the statistical mean or mode of all these individual estimates. In the event of big disagreements, the estimates are discussed and the task is more elaborately analyzed, followed by another round of planning poker. The estimation is over when all developers agree on the plenary estimate.

5. Time-boxed sprints producing potentially shippable output
The sprints are time-boxed to a fixed length, often two, three or four weeks. A new version of the software is released at the end of a sprint.

6. Mutual commitment to Sprint Backlog between Product Owner and Team
The whole team, developers and Scrum Master, and Product Owner is involved in the sprint backlog and mutually committed to complete the tasks therein.

7. Short daily meeting to resolve current issues
A brief, informal meeting is arranged every day. Each developer describes very shortly what he/she has done the day before and what the further plans are. Any new challenges or issues uncovered are discussed. These meetings are also known as stand-up meetings, as they often are conducted in standing position in order to keep them short.

8. Team members volunteer for tasks (self-organizing team)
The developers choose freely from the tasks in the sprint backlog. There is no delegation of work/tasks.

9. Burndown chart to monitor sprint progress
The team creates burndown charts (see below figure from [31]) to keep track of the progress. The burndown chart displays how much work is done according to time. The chart often has several lines. Almost always included is the line of actual burndown (remaining effort in the below figure) and ideal burndown (linear line).

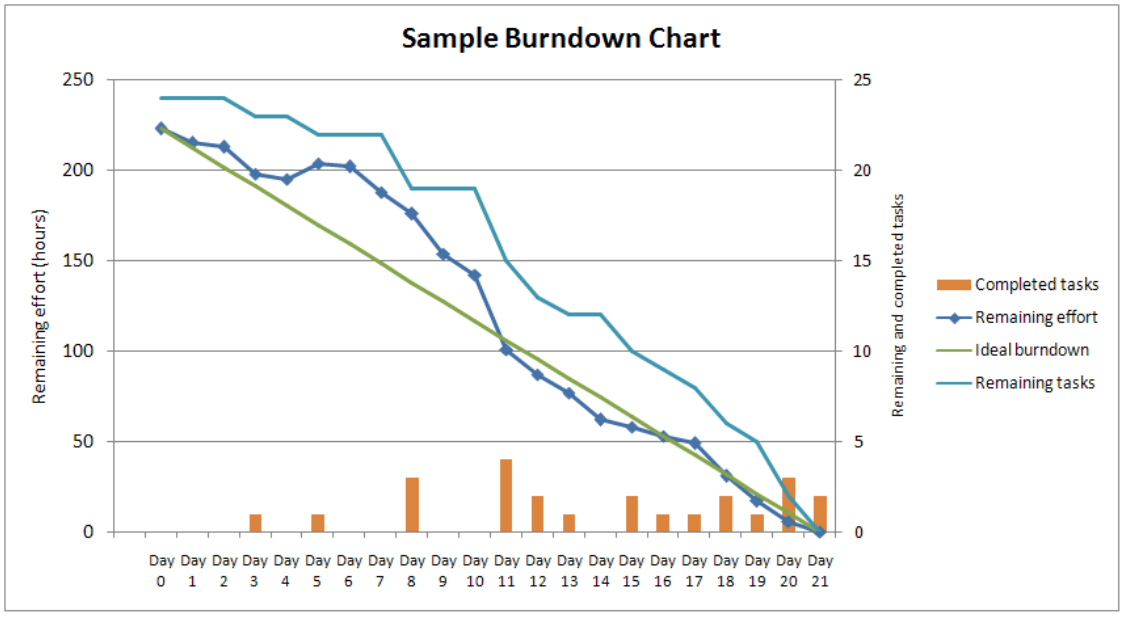

Figure 1: Example burndown chart

10. Sprint review meeting to present completed work
In the final stages of a sprint a review meeting is arranged where customers, stakeholders and other interested people are invited. The agenda for the meeting is to present the tasks completed during the sprint.

11. Sprint retrospective to learn from previous sprint
The sprint retrospective meeting is arranged in the final stages of the sprint. The development team looks back on practices and general issues which emerged during the course of the sprint. Both good and bad practices/issues are subjects for discussion. Improvements are being suggested, and the process is altered accordingly.

12. Release planning to release product increments
Product increments (sprint backlogs) are planned in release planning meetings. These can be planned quite accurately using the data from burndown chart and measurements of project velocity (how much gets done in a single sprint).

13. User stories are written:
In order to define tasks user stories are created. User stories have be written according to a certain pattern, along the following lines:
"As <customer, role, stakeholder>, I want <requirement, goal, desire> so <purpose, motivation>"

14. Dedicated open workspace
In order to facilitate communication and collaboration, the team should be situated in an open workspace environment.

15. Sustainable pace
The development pace should be sustainable. The work planned for the release must be possible to complete without working overtime or making considerable cuts in functionality and quality.

16. The project Velocity is measured
The project velocity is a measure of how much work gets done in a sprint. This measure is used when planning releases, to determine the amount of (estimated) work that can be included in the iteration.

17. Move people around
Developers are supposed to work with different parts of the code. This is done to prevent that developers become “experts” in certain parts of the software and to ensure that every developer is able to work on most parts of the system.

18. The customer is always available
A customer or a representative for the customer is available at all times, in the event that issues needing further input or clarification should arise.

19. Code written to agreed standards
Code must have consistent formatting and standard.

20. Code the unit test first
Define test cases prior to writing the code. The initial test cases will naturally fail, but it will give the developer a clear picture of what needs to be done in order to complete the task, as well as providing a constant feedback loop.

21. All production code is pair programmed
Pair programming means that two developers share the same computer when writing code. The practice ensures good quality code and constant review.

22. Only one pair integrates code at a time
There must at no times be more than one pair of developers making changes to the source code repository. This is performed to ensure an error-free integration, yet allowing parallel development of software.

23. Integrate often
Changes should be uploaded regularly (at least once a day), as long as the code works and are written according to agreed standards.

24. Set up a dedicated integration computer
This is done primarily to facilitate practice 22.

25. Use collective ownership
The developers should feel a strong, collective ownership to the software, so that everyone can possibly contribute to all main aspects of the project.

26. Simplicity in design
Choose a simple design, suitable for solving the task at hand. There is no need to design for future enhancements, as these are not defined yet and such advanced design will only lead to more complex code than necessary.

27. Choose a system metaphor
To be able to easily explain the system design to customers and new people in the project, an accessible metaphor for the system must be stipulated. The metaphor is essentially a logical representation of the architecture.

28. Use CRC cards for design sessions
CRC is an abbreviation for “Class, Responsibilities and Collaboration”. These CRC cards are used by the whole development team in design sessions, for evaluating the relationship and messaging between objects and classes. The purpose of applying the technique is to easily collect and evaluate design ideas.

29. Create spike solutions to reduce risk
A spike solution is a program aiming to address very hard technical or design-related problems. Such programs investigate and assess potential solutions.

30. No functionality is added early
Functionality will not be developed until it is needed. Adding extra flexibility, regardless of how it improves the system, will slow down the development and perhaps also be a complete waste of time as it may never be required.

31. Refactor whenever and wherever possible
Change the architecture and design whenever it can be replaced by something better or more appropriate. Requirements can change quickly, which mean that architecture and design must follow. Constant refactoring is an important action in order to ensure the software's maintainability and simplicity.

32. All code must have unit tests
A unit test is an assertion, or a set of assertions, about a component of the software, for instance a single method or function, which either passes or fails. All code must be tested by dedicated unit tests.

33. All code must pass all unit tests before it can be released
Nothing can be released if even a single unit test fails.

34. When a bug is found tests are created
New tests, addressing the discovered bug, are written and added to the test suite.

35. Acceptance tests are run often and the score is published
Acceptance tests are tests created based on user stories and is a type of black box testing (testing the experience of the program, without knowing the details of the implementation). All acceptance tests must be passed in order for a story to be resolved and complete.

4.5.2 Omitted practices

Six XP practices from [7] were covered by the Scrum practices. Due to this overlap, the practices are therefore intentionally omitted from the agile mapping chart (table 4). The practices, however, are still addressed in the agile mapping chart (as they are covered by equivalent Scrum practices). The six omitted practices are:

EP1. Release planning creates the release schedule.
EP2. A stand up meeting starts each day.
EP3. Make frequent small releases.
EP4. The project is divided into iterations.
EP5. Iteration planning starts each iteration.
EP6. Fix XP when it breaks.

EP1 is covered by practice 12. EP2 is covered by practice 7. EP3 is covered by practices 5 and 12. EP4 is covered by practice 5. EP5 is covered by practice 3. EP6 is covered by practice 11.

5 Systematic literature review on agile practices in scientific software development

In this section, a systematic literature review investigating agile practices in scientific software development is presented. The motivation for carrying out the review is discussed in section 5.1. Thereafter, the systematic review research method and how it was used is presented. Each of the finally included papers is then presented and their quality assessed. In section 5.4 the agile mapping chart for all the relevant articles (or projects described in the articles) is presented. The chapter closes with a synthesis of the general trends and results observed in the studies.

5.1 Motivation and research propositions

The literature review is centered on commonly encountered challenges in scientific software projects, one of the most widely-recognized of which is testing. Several articles [2; 3; 8] mentioned difficulties and inadequate handling of testing, validation or verification of the software. Requirements handling was also an aspect accentuated as challenging.

The purpose of the literature review is two-fold: First, to investigate the extent to which agile practices have been used in scientific software projects. In the second phase the eventual impact (of agile practices) on testing and requirements activities in these projects is investigated. As described in section 1.2, the propositions related to research question 2 are the following:

> **P1**. Projects using agile practices have a better handling of testing-related activities.
>
> **P2**. Projects using agile practices have a better handling of requirements activities.

5.2 Research method

The literature review is performed in a similar fashion to the method described in [19]. Due to the sheer number of research fields where scientific software development can be found, multiple literature databases had to be included in order to ensure that a sufficiently comprehensive result set would be returned.

The search query was a conjunction of the following sub-queries:

Q1. XP AND scientific AND software
Q2. Agile AND scientific AND software
Q3. Agile AND scientific AND research

Q4. XP AND scientific AND research
Q5. Scrum AND scientific
Q6. Crystal AND scientific

Consequently, the pattern of the complete query was: ‘Q1 or Q2 or Q3 or Q4 or Q5 or Q6’.

Agile terms like *lean development* or *feature-driven development* could have been included in the search queries. However, research on less renowned agile methodologies (compared to Scrum and XP), would likely be covered by either Q2 or Q3.

The query yielded a great number of results; some databases provided a three-figure number of hits. Most of the papers were clearly irrelevant; many of them originated from Q6 and were related to chemistry research into objects with crystalline structure. A large proportion of clearly irrelevant papers also described the apparent lack of scientific foundation for agile practices. Some papers on how to execute scientific software on the Windows XP platform also proved to be irrelevant.

The papers were collected from the ACM, IEEE Xplore, ScienceDirect and ISI Web of Science databases. After searching the databases, the same keywords were run through Google Scholar in order to collect any relevant papers falling short of the original search. This search identified one additional paper (number 9 in the list in Section 5.3). Statistics of the literature search and subsequent filtering are presented in table 5; the number of actual results (for each sub-query), total number of unique articles and statistics on how many articles were excluded, based on title and abstract respectively, are included along with the number of relevant studies for each of the databases.

A large number of the retrieved papers could be excluded based solely on the title. This was the case when aspects other than software development were the main focus; for example when the paper did not portray software development at all, or when the science behind rather than the development of the software was reported. If the title did not exclude the paper, the abstracts were thoroughly examined. In the IEEE Xplore and ISI Web of Science databases, some fine-tuning of sub-query 6 was necessary due to an overwhelming amount of results (due to the above mentioned chemistry papers). This refinement was based on publication year (papers published prior to year 2000 were filtered out), as well as on filters for publication title and subject provided by the respective database search engines.

Table 5: Summary of search results and filtering

	ACM	IEEE	ScienceDirect	ISI Web
SQ1	5	7	11	11
SQ2	14	21	3	24
SQ3	12	18	4	27
SQ4	3	1	1	7

SQ5	2	3	1	3
SQ6	26	3014	305	1004
SQ6 (refined)	26	114	305	4
Total unique	49	145	320	59
Excluded title	36	133	313	59
Excluded abstract	11	8	6	42
Relevant	2	4	1	6
Total unique relevant	**8**			

5.3 Relevant papers

The literature search, and subsequent filtering, resulted in the following list of papers eligible for full review:

1. Engineering the Software for Understanding Climate Change [32]
2. An empirical characterization of scientific software development projects according to the Boehm and Turner model: A progress report [33]
3. Test driven development and the scientific method [34]
4. Chaste: using agile programming techniques to develop computational biology software [35]
5. Agile methods in biomedical software development: a multi-site experience report [36]
6. When software engineers met research scientists: A case study [16]
7. Exploring XP for scientific research [37]
8. Is Scrum and XP suitable for CSE Development? [5]
9. Introducing Agile Development into Bioinformatics: an Experience Report [38]

After examining the above papers in detail, four of them (number 2, 3, 6 and 8 in the list) could be excluded from any further review: Paper 2 focuses on a future, planned study where the objective is to obtain an empirical characterization of scientific software, one of the aims being to assess the suitability of agile and plan-driven approaches to scientific software projects. The proposed study, when completed, will however be relevant for the same line of research as this thesis. In paper 3, the techniques and practices of XP are compared to the manner of conducting scientific inquiries. Some similarities are investigated (for instance how test-driven development resembles theory building and exploration), but the study is not

directly related to scientific software projects and the applied processes therein. A scientific software project was described in paper 6, but the applied process was plan-driven. Due to project issues and largely unsatisfactory development, the authors discuss whether agile practices could be introduced and whether these would to some extent resolve the problems they encountered. Paper 8, focused on whether it is sensible to use agile practices in scientific software projects. They investigate the constituents of agile methods and assess how each of them aligns with the desiderata of scientific software development. None of the four above mentioned papers reported on any real experiences with using agile practices in scientific software development projects, and were therefore excluded from any further review.

Summaries, with particular emphasis on validity and relevance, of the remaining five papers are presented in the following sub-sections.

5.3.1 Paper 1 – Engineering the Software for Understanding Climate Change

"Engineering the Software for Understanding Climate Change" presents the results of a case study investigating the development practices exercised by climate researchers at the Met Office Hadley Centre. The paper is a collaborative effort between a climate scientist from the research center and a software engineer from the University of Toronto, Canada. Empirical evidence was collected from 24 interviews with participating scientists, and from direct observations of meetings/workshops, and quantitative data were extracted from the code base.

The aim of the study was to investigate the current practices employed by the scientists at Met Office Hadley Centre. The high degree of agility present in the development process was surprising to the software engineer. One of the most significant differences, when compared to other scientific software projects, was the emphasis put on verification and validation activities. The authors also claim that requirements activities followed a (semi-) agile approach. As there was no explicit agile method enforced in the project, the high level of agility identified was in itself an important result.

The authors discuss some of the study's validity threats, and the actions undertaken to handle these. One of the threats mentioned is terminology issues; certain terms are not easy to discuss in the interviews, as the scientists may have a different understanding and recognition of software engineering terms and concepts. Follow-up interviews and feedback sessions with the interviewees were organized to reduce this threat.

The research questions for this study were not directly related to effects of agile practices. Agile practices were employed, but no explicit agile method was used, and it is difficult to know whether aspects of agility in the process or other factors, such as the level of correctness required in the domain of climate change, caused the good testing practices. Also, the development process had some discrepancies regarding the use of agile practices (see table 6), making it questionable whether testing activities indeed were executed in an "agile" manner.

As the authors of paper 1 report, there are some external validity issues. There was only one project under examination. Although it might be a representative case for the specific domain of climate change, the project does not necessarily represent scientific software in general. Some alternative explanations are then plausible, as is also indicated in the paper; the way in which the project adapted and tailored agile practices might have been influenced by the climate change domain's testing requirements, which may not be as strict in other domains.

The connection between presented evidence and the claims/results is not explicit. This may be due to the paper aiming to characterize the development practices found in the project. Its main projective was not to assess the suitability of agile methods, and even less to emphasize on the effects of an agile approach. Therefore, the paper is mostly relevant for analyzing the presence of agile practices, and to a lesser degree suited to examining the effects of such practices.

The project is referred to as Project 1 in table 6.

5.3.2 Paper 2 – Chaste: Using Agile Programming Techniques to Develop Computational Biology Software

Chaste is a computational biology project with a large number of scientists involved. The aim of the project is to provide a library for cardiac modeling and cardiac electrical activity simulation. The paper is written by a total of ten researchers, stringing together efforts from both computer scientists and biologists. A case study regarding the use of agile methods is the topic of the paper.

The introduction of XP into the Chaste project was claimed to be a massive success. They found the basic agile principle of being responsive to change to be very much in the natural spirit of general scientific research. Consequently they favored the responsive ability imposed by adopting XP in the project. The authors also emphasized that the agile approach to testing was a valuable asset, concerning both the testing of new functionality and regression testing of existing functionality.

The evidence is presented in a reasonably comprehensive manner, although organizational aspects, such as the composition of teams, are not described in much detail. The structure within and across the teams, as well as the number of teams and members within a team, remain unknown. It is suggested that there was a large number of scientists involved and it would have been interesting to know more about the organizational aspects in order to be able to more thoroughly discuss the potential for generalizing the results from the paper.

The project is referred to as Project 2 in table 6.

5.3.3 Paper 3 – Agile Methods in Biomedical Software Development: A Multi-Site Experience Report

Paper 3 is another study in the field of bioinformatics. The paper reports on experiences from multiple sites and projects. A total of six projects, all incorporating key agile practices, are examined by the authors. The multidisciplinary group of authors represents different universities and research centers, all based in the United States.

Agile methods were deemed very suitable to biomedical software development. The developers regarded the agile approach to be a key success factor. In this line of software development, the software has to be responsive to change at two levels; progress in the scientific domain as well as specific customer demands may both enforce changes to the software.

Subjective experiences are the primary source of evidence in the paper. The group of projects under examination was selected during meetings and biomedical conference discussions. To collect and elicit the tacit knowledge and experience from the involved parties, the authors initiated a basic mapping survey. Thereafter they conducted open-ended interviews with key developers in the projects. To ensure the quality of the collected data, several rounds of feedback sessions were arranged. The authors extensively described the evidence and the method of evidence collection.

A notable advantage compared to the other identified studies is the fact that data was collected from six different projects. Similar effects were reported in all cases, strengthening the claims of positive effects due to agile practices.

However, the cases under examination have some obvious similarities, restraining the scope for external validity:

1. They were all of small size (a single team with 2-5 members)
2. They were all in the domain of biomedical software development.

The first issue is consistent with the notion that small size projects are more inclined to succeed with agile methods than larger projects (with multiple teams). It has been suggested that XP does not scale well to extensive projects [11]. The second issue pertains to whether some common attributes present in biomedical software development make them more prone to embark on and succeed with agile development methods. It would be interesting to observe whether a contrast case shared the same results as the ones investigated.

The six projects described in the paper are referred to as Projects 3.1 to 3.6 in table 6.

5.3.4 Paper 4 - Exploring XP for Scientific Research

In this paper the authors reported on an attempt to apply XP to a project at a NASA research center. The two authors worked closely together, and were the only developers involved.

They aimed at assessing the suitability of XP to a scientific software project, and reported that XP was successfully adopted. More specifically, code quality improved, more bugs were caught, development was more focused, maintenance was easier and productivity increased.

Although the use of XP was reported as promising, there are some limitations to the study's generalizability. Firstly, the software project was very small, consisting of only two members. Furthermore, some of the applied practices, such as pair programming, are only possible to perform when the scientists are co-located and available concurrently, which is not always the case. The paper is still relevant to small scientific project teams, but the results might not be transferable to scientific projects of larger size.

Another factor limiting the project's relevance to the scientific community at large is that the developers could work exclusively with this project during the two-month life span of the project. In terms of generalizability, there are two problems with this situation: First, most scientific software projects have a life span of several years, sometimes even decades, meaning that any long-term effect of using XP cannot be addressed by this study. Second, even though an increasing part of work time is devoted to writing software, most scientists are engaged in other research and education activities. Opportunities for full-time dedication to development for all team members in a (large) scientific software project are scarce. The project context remains somewhat remote from the regular settings of scientific software projects.

The researchers had no experience with agile methods prior to the experiment; no software engineers were involved in either planning or execution. The assessment of how well the new development methodology was implemented is, consequently, subjective. The authors, although being excellent researchers and capable scientific programmers, still lack software engineering knowledge. The reader might question the extent to which agile practices were carried out properly, as well as the validity of the reported results.

Another aspect worth noting is the fact that the effects of applying XP may be confounded with effects of other new elements introduced in the project. Ruby and object-oriented design represented two completely unfamiliar concepts for the two scientists/developers. The two researchers' usual language, Fortran, is quite different from Ruby. With its object-orientation and testing frameworks, Ruby might have been as significant to the alleged improvements observed by applying XP. Consequently, the reasons for the researchers' apparent preference of XP to prior practices may be related to the other elements introduced (or a combination of these and XP). No substantial evidence to indicate that the testing improvements are exclusively caused by the use of XP in the project is presented in the report.

The project is referred to as Project 4 in table 6.

5.3.5 Paper 5 - Introducing Agile Development into Bioinformatics: an Experience Report

One of the authors of Paper 3 [36] also wrote a report on the experiences of introducing agile techniques in a bioinformatics project. This project was not among the six projects described in Paper 3. The author presents an application of an agile method to their process, as an answer to their need for being more responsive to changing requirements. Various agile practices, adopted from a combination of Scrum and XP, were then incrementally incorporated into the development process.

The authors report positive experiences with implementing agile practices in increments, focusing on testing and requirements activities. They found the agile practices to be beneficial in dealing with flexible requirements. The agile testing practices also facilitated the scientific setting where correct and reproducible results are of the utmost importance.

The evidence consists of the subjectively reported experiences of the involved project members. The incremental fashion of introducing the agile method is well documented in the paper, while the effects of each increment are documented to a lesser degree. It is therefore difficult to find evidence on the effects of agile practices. Consequently, the findings in this study cannot be particularly emphasized when conclusions are made in the synthesis.

The project is referred to as Project 5 in table 6.

5.4 Mapping of projects to agile practices

Once the final set of relevant papers was defined, a yes/no indicator was used to evaluate each reported project against each individual practice in the agile mapping chart. Table 6 shows the result of this mapping. Fields are left blank if it was not possible to determine, from the available information, whether a practice was followed. This was particularly the case with papers more focused on the effects of the agile approach, rather than naming or describing the employed practices in much detail. Even in such papers, it was nevertheless possible to ascertain whether certain practices were followed.

Table 6: Agile mapping for the examined projects

	Projects									
#	*1*	*2*	*3.1*	*3.2*	*3.3*	*3.4*	*3.5*	*3.6*	*4*	*5*
1	No								No	
2	No								No	
3									No	
4	No								No	
5		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
6	No								No	
7	No	Yes	Yes			Yes	Yes	Yes	Yes	Yes
8	Yes		Yes	Yes	No	Yes	Yes	Yes	Yes	

9	No	Yes							Yes	
10									No	
11									Yes	
12	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
13	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes		Yes
14	Yes		Yes	Yes	Yes	Yes	Yes	Yes	Yes	
15									Yes	
16	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
17	No	Yes								Yes
18	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
19	Yes								Yes	Yes
20		Yes								
21		No	No	No	No	No	No	No	Yes	
22									Yes	
23	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
24										
25	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes	Yes
26									Yes	
27									Yes	
28										
29										
30	Yes									
31		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
32		Yes							Yes	
33										
34		Yes								
35	Yes	Yes						Yes		

For most of the practices evaluated to “no” in the table, it was clearly stated in the articles that the practice was not followed. In a few cases, the articles presented evidence that clearly contradicted the presence of one or more practices. An example of the latter situation was found in project 1. Practice 1 (*Priorities (Product Backlog) maintained by a dedicated role (Product Owner)*), was not used, because the paper stated that all project members identified, specified and prioritized new features, hence it could be deduced that prioritization was not a centralized responsibility of an eventual Product Owner. Due to this individual requirement handling, practice 4 (*Planning poker to estimate tasks during Sprint Planning*), and practice 6 (*Mutual commitment to Sprint Backlog between Product Owner and Team*) could also be evaluated to not present.

5.5 Synthesis of findings

The identified papers were all experience reports, presenting evidence typically gathered from interviewing key project members. Some authors also used direct observations and multiple feedback interview sessions [32; 36]. The authors of the papers were often participants in the systems development themselves, relying on a combination of their own expert opinion and personal experience when arguing for their claims. This may cause both researcher and personal bias, especially in papers with few authors. Some of the studies [32; 38; 36] did not have the effects or results of applying agile methods as the primary focus. The claim/evidence relationship is less obvious in these studies than in studies focusing on applying an explicit agile method and reporting on the results of doing so [35; 37].

The papers all indicated positive effects of agile practices in scientific software developments, indicating that agile methods may indeed effectively handle the special characteristics of requirements and testing in scientific software development. The evidence in favor of such a conclusion is stronger for small projects with relatively few team members.

Almost all of the studies reported on improvements to the testing activity. Testing was performed more extensively, and the approaches to adding tests of new modules improved [35; 37] .The rigor of the testing approaches seemed to satisfy the need for having reproducible, correct results [38]. For requirements activities, a certain degree of mismatch was identified between scientific software projects and the agile-assumed context of a clear customer/developer relationship. However, the responsiveness and flexibility of agile methods proved valuable for the requirements activities. Elicitation and specification of tasks were perceived as easier and more focused with agile methods [35; 37]. Good practices regarding requirements prioritization were also observed [32].

In conclusion, the literature review supported proposition P1, i.e., that projects using agile practices have a better handling of testing-related activities. The review partly supports proposition P2, i.e., that projects using agile practices have better handling of requirements activities, though the findings are not as substantial. In that respect, it is also worth noting that requirements activities were not necessarily considered to be a significant problem for small-size projects [8].

6 Case Study

This section presents the case study. Section 6.1 contains descriptions of the general design of the case study, the research questions and the rationale for choice of research method and data collection (interviews). Section 6.2 presents the cases, while section 6.3 describes the interviews in greater detail, along with the associated risks of using this type of evidence source and the measures undertaken to reduce the effect of these risks. The results from the case study are presented in the next chapter.

6.1 About the case study

The case study focuses on three different, active scientific software projects. The projects are all international collaborations, but the interviewees work in sub-projects located in Norway. Key developers from each project are interviewed, in order to gather the necessary evidence for an agile evaluation.

The purpose of the case study can be formalized into the three following phases:

1. Analyze and conceptualize core process elements in the software development processes in the three projects.
2. Address research question 1: Investigate to what extent these elements map to practices in agile methodologies, i.e. evaluation according to the agile mapping chart in table 4.
3. Address research question 2: Analyze the effects of the agile practices.

In each project, 2 to 4 key developers were interviewed. Access to the projects was arranged through the network of one of my supervisors.

6.2 Cases

Key characteristics of the cases are presented in table 7. The projects are all large, established projects with many active participants. Although the project domains are all within the realm of natural sciences, they differ somewhat. Apart from the domain, there is also diversity in the other characteristics, regarding choice of process model, commerciality, programming language, project duration and number of developers.

Table 7: Case characteristics

	FEniCS	Dalton	Olga
Scientific domain	Mathematical (Automated solution of differential equations)	Chemistry (Molecular electronic structures)	Physics (Flow modeling of oil, gas and natural water)
Number of developers	> 10 [1]	40 [2]	50
Duration	10 years	30 years	30 years [3]
Programming languages	C++, Python	Fortran77/90, C, C++	Fortran, C++, C#
Chosen process method	No specific	No specific	Scrum
Distributed development	Yes	Yes	Yes
Availability	Free, open source	Free, licensed	Proprietary

[1]10 members of the "core team", the project is open source; number of developers is greater than 10, but the exact, actual number vary somewhat.
[2]Approximate number of participants involved in the course of the project, not all of which are currently active.
[3]Initial development started in 1980, first commercial release in 1991.

6.2.1 FEniCS

The first case under investigation is FEniCS, a mutual software project joining together participants from several universities and research institutions in the domain of computational mathematics. The aim of the project is to facilitate automatic solution of differential equations. Although the software is in a constantly operational state, no explicit version 1.0 of FEniCS has been released (at the time of the interviews). The program is open source and available for everyone. It is even distributed through software managers in Ubuntu and Debian (popular Linux distributions).

FEniCS is no traditional software application, but rather a collection of (more or less) separate packages that form a framework. Researchers write applications, typically related to their specific scientific area of interest/expertise, on top of this framework/interface. FEniCS components are written in C++ and Python. An international community of developers contributes with coding and documentation. Thus, the development is fairly distributed.

6.2.2 Dalton

The Dalton project is an older scientific software project, in the molecular electronic structures sub-domain of chemistry, aiming to automate computation of such molecular properties. The software was first released in 1997, with additional versions in the years to follow; the latest version, at the time of writing, was released in the first quarter of 2010. There is an international community of scientists involved in the development of the program. Most of the main actors are located in Scandinavia.

The program is primarily written in Fortran 77 and Fortran 90. Some components are written in C. The authors recommend a UNIX platform for the software. The program consists of seven components, with more or less independent development cycles. The program is distributed free of charge, as long as the user signs a personal license agreement. Site licenses of Dalton are also available.

6.2.3 Olga

The third case is Olga. Contrary to the other cases, this is a commercial project and the software is developed by the SPT Group. Olga is a simulator tool for accurate flow modeling of oil, water and gas in wells and pipelines. Being a commercial system that needs to stay competitive, Olga has a more defined customer segment and a more direct customer-developer-relationship than the other two projects in the case study.

This project is also distributed, with offices in several countries. These offices cooperate closely in the development. There are approximately fifty developers in total, which participate in various subprojects. The programming languages are Fortran, C++ and C#. The main core of the program is programmed in Fortran and is rather established and stable. Other parts of the software are written in either C++ or Fortran, while the graphical user interface (GUI) is primarily coded with C#.

6.2.4 On the selection and the representativeness of the cases

The three cases will complement the projects investigated in the literature review. They will represent different types of scientific software than the projects investigated in the review, as they, generally, are much larger in terms of size, life-cycle and participants. By the selection of cases, multiple domains – and domains other than bioinformatics – are investigated. In the event of detecting agile practices in the cases investigated, the combined analysis of these and the projects examined in the literature review will enhance the evidence base, and possibly, increase the potential for generalization regarding research question 2 (about the effects of agile practices in scientific software development).

The contexts and general conditions of the projects are pretty representative (at least when compared to the general characterization of scientific software development in chapter 2), but exactly how well the development processes found therein match those in the general characterization of scientific software remains to be uncovered. An assessment of how representative these projects are will be provided in the later discussion (chapter 8) in order to evaluate the external validity of the case study results.

6.3 About the interviews

In each project 2 – 4 key developers are interviewed. The interviews were recorded with the consent of the interviewees. The interview sessions lasted approximately 1 hour, sometimes a bit longer. The interviews were conducted on an individual basis, with multiple interviews per project. There were a few exceptions; two developers were present at the second Dalton interview and the second FEniCS developer was interviewed over two separate sessions (one session about the general aspects and one session about the agile mapping chart). Also, the agile mapping chart was not reviewed with the first developer in FEniCS, as he was currently not active in the development of the project.

The interviews, aiming to address multiple aspects of the development process and to determine the eventual presence of agile practices, were semi-structured. The interview guide consisted of two rather separate parts; one general and open-ended part where the interviewee to some degree could determine which aspects was elaborated in detail, whereas the other was more structured, seeing as its purpose was to review the 35 practices in the agile mapping chart (table 4) one by one.

The questions in the first part focused on key aspects and activities within the software development process. Even though it was up to the interviewee (at least to a certain degree) how heavily each aspect was emphasized, the discussion of following key aspects was ensured to be covered in all interviews:

- General overview of the development process, including:
    - Activities and eventual relationships between them
    - Roles of the project members
- Specific activities or aspects of the development, including:
    - Testing practices
    - Requirements (elicitation, specification, estimation and prioritization of these)
    - Customer/developer relationship
    - Code design and maintenance
    - Presentation of the work

In some interview sessions, it became apparent early on that some aspects were of limited interest to the developer or of limited relevance to the project at large. For instance, trying to identify or formalize the activities in the two non-commercial projects was especially difficult, as the development is very individual. This makes it hard both for me as an interviewer and for the interviewee to conceptualize the descriptive process. The initial plan for the second part of the interview was to question the interviewee about each practice from start to end. To do this, I had to provide enough information for the interviewee to evaluate the specific practice. Thus, a variable amount of time in each session was devoted to the explanation of agile practices.

The interview guide did change somewhat along the way, as I listened through the first interviews conducted; the original strategy had to be altered as the second part of the interview consumed too much of the designated interview time (due to both the sheer number of practices and the amount of explaining needed to make sure that the interviewee understood the practices, especially Scrum ones, sufficiently). Another argument for not going through every single practice in detail was the fact that some of the aspects discovered during the first, general part of the interview clearly contradicted the presence of some of the agile practices. For instance the practice of using planning poker to estimate the tasks in the sprint backlog may be contradicted by two factors; the absence of estimation effort all together or the absence of any equivalent to a sprint backlog. Additionally, the absence of a specific practice may render the presence of other practices impossible. An example here is practices 1 and 6.

A decent compromise, which were performed in some of the interviews, was to focus on the aspects which were emphasized to a small degree in the first part of the interview, and then focus on the agile practices in the mapping chart where additional input was needed to obtain a correct impression of the practice and determine whether it was present or not in the project. The nature of the specific agile practice also played a part here; some of the Scrum practices were left out, as their eventual presence in the project was essentially ruled out by the first part of the interview. However, in most of the interviews we were able to discuss virtually all the agile practices.

7 Case study results and analysis

This chapter presents the results from the case study. First, the individual interviews from all projects are presented in order to identify the general development process, its activities and roles. Requirements and testing activities are given explicit attention, as is eventual other aspects emphasized by the interviewees. Perceived challenges are also presented. After the presentation of individual interview results, the summarized, general results from the three scientific software projects are presented.

Secondly, there is a section (7.4) summarizing the results from the agile mapping chart for the projects. This section is based on the combined impressions from the different interviews in each project. Disagreements between the interviewees are discussed and considered in order to determine whether each practice was present or not. Eventual similar, but not completely alike, practices are discussed and presented as well.

7.1 FEniCS

Three developers were interviewed in the FEniCS project. All of the interviews were conducted individually. The first and third interviews were conducted in a single session. The second developer was interviewed over two separate sessions.

7.1.1 First interview

7.1.1.1 About interview/interviewee

This developer has been one of the main developers in several of the FEniCS components, but is currently not as active in the development. Although some courses in programming, primarily oriented towards algorithms and data structures, have been part of the education, the background is primarily oriented towards mathematics. Thus, the interviewee had no formal training or education in theoretical aspects of software engineering.

The agile mapping chart was not established at the time of this interview. Since the developer was not as active and due to pending interviews with more actively involved members in the project, an additional interview session focusing on the agile practices was not arranged.

7.1.1.2 Teams and roles

The distributed teams in the FEniCS projects are very ad-hoc and the team members organize themselves in the way they see fit. There are no team leaders in the traditional sense, and no delegation of tasks. An exception from this rule is perhaps when a researcher assigns (very) large tasks to a PhD student. However, it is most common that each developer chooses his/her own level of commitment and tasks to tend to. It is hard to identify any specific roles, but one group may be referred to as “main developers”. This is not a formal role, but it demonstrates

that there are some differences with regards to the level of a developer. This level is based on his/hers contributions to the project. The main developers are very central and perform quality assurance on a lot of the code.

7.1.1.3 Development process

The development is incremental, but there are no stipulated milestones in the project, perhaps with the exception of a planned version 1.0-release (which has a set date). So far, releases have followed some of the Ubuntu releases. The interviewee found it hard to identify or name any of the activities in the development process, as the primary motivation for development is personal initiative and commitment. There are many developers with different needs and focus, which “pull” the development in different directions. Each individual developer takes care of the prioritization of tasks, as there are no plenary prioritization activities.

7.1.1.4 Requirements and testing

The project uses LaunchPad to handle requirements, to track bugs and for overall project organization. A traditional customer/developer-relationship is rare. It is possible to define and register blueprints (i.e. tasks) in LaunchPad, but there is no guarantee that it will be tended to, as developers only implement the blueprints they want or need themselves. Tasks are generally not estimated. There is no specific set of phases, but a common approach may be something like the following: First there is some initial planning. Then, the developer wanting the functionality assigns the task to himself/herself and implements it. After implementation, some testing is performed and a main developer checks the code before it is finally committed to the main code repository.

Automatic tests ensure that the project builds - that existing tests passes, and that the code compiles. However, testing activities, or other activities for that matter, does not represent a discrete phase in the development. The developers are not required to provide test cases for their code; this is completely up to each person. From time to time, code that breaks the tests is committed, but this does not represent a big problem in the development. Generally, there is no particular focus on testing; implementation is the most important activity.

7.1.1.5 Other aspects

The most important way of communication is open mailing lists, due to the distributed nature of the project. There are also some other communication channels, such as personal meetings and yearly conferences. Personal meetings are important when a group of developers are co-located. Changes in requirements are handled via mailing lists and/or in LaunchPad. Much work is currently (at the time of the interview) being put into writing and organizing the project’s documentation, primarily user documentation. In addition, there are certain pieces of technical documentation, such as reference models and code comments. Personal commitment is significant for such efforts.

7.1.1.6 Challenges

Challenges may arise when users/developers of the software have different demands regarding the accuracy of the software. As there are potential overlaps in science, the same functionality may be used by researchers from different domains.

7.1.2 Second interview

7.1.2.1 About interview/interviewee

The second interviewee's background is similar to that of the first one. The limited formal training in computer science consists of courses on data structures and algorithms, with a clear focus on programming. These courses did not devote much attention to managerial or organizational aspects of software development. The interviewee is a key member of FEniCS and has been central in the development for quite some time.

7.1.2.2 Teams and roles

There are no defined or very clear roles in the project. However, the interviewee points out that some roles have emerged during the course of the project. Each sub-project has its own "core team", and only members of this team have access to commit code into the main code repository. Code changes have to go through one of the main developers before it is accepted. Membership in the core team has to be approved by a project administrator.

Personal involvement and interest is identified as the key motivation factors, as the software is used extensively in relation to your own research. Developer and user/customer is often the same person. Although the personal initiative is hugely significant, there are still occasions where developers have to coordinate and work together to find a solution. This may happen when extensive or complex modifications are due or when there is an overlap of interest regarding underlying science or functionality.

7.1.2.3 Development process

The interviewee found it hard to identify any specific phases or activities in the process. It is pointed out that there sometimes is an analysis step (but not for every task) – perhaps an assessment of the mathematics involved and whether or not the task is possible to implement in FEniCS. The interviewee sometimes uses a "demo-driven" approach where a prototype or a demo is written prior to implementing the task.

7.1.2.4 Requirements and testing

The project uses LaunchPad to handle aspects like requirements, bug tracking and code repository. Blueprints (i.e. tasks) are created for planned features, which are prioritized by the developer requesting the functionality. Priority may also be set by a member of the core team. Blueprints can be discussed either via LaunchPad or in the open mailing lists. There is no estimation of the tasks/blueprints at all.

The amount of preparation, analysis and discussion is closely connected to the specifics of a task. If the idea behind a task is a bit unclear it is not necessarily inserted into LaunchPad as a blueprint, “although that might have been a good idea”; it is inserted when the authors of the task are confident that it is possible to implement it. Some tasks, typically small maintenance tasks, are not necessarily registered as blueprints.

How to perform testing activities is up to each individual developer. Testing activities are not really a big focus. There are some regression tests (from demos) and unit tests which cover some parts of the code. There are also automatic tests.

7.1.2.5 Other aspects

Open mailing lists is regarded as the most important communication channel. LaunchPad is used for requirements and bug handling, as well as Q&A where users may ask questions about the software and its packages. Yearly FEniCS workshops or conferences are arranged. At these gatherings new parts of the software may be presented (the presentations are mostly oriented towards results obtained by using the software, rather than the software itself). No professional software engineers are involved in FEniCS. The participants are scientists with an interest in the software and the results it produces.

7.1.2.6 Challenges

There have been some challenging aspects in the development, especially with regards to bug tracking. The developers found it very hard to administer bugs, which originally were organized in a bug tracking tool or in the mailing lists. Bugs had a tendency to “disappear/drown” in these communication channels. Tracking these bugs became more and more challenging, but the situation improved greatly after LaunchPad was introduced.

7.1.3 Third interview

7.1.3.1 About interview/interviewee

The third developer is perhaps the most influential member of FEniCS, and certainly one of the most active participants. This developer has been a member of the project for nine years, ever since its initiation. The interviewee is pretty much an autodidact when it comes to programming, software engineering concepts and project work.

7.1.3.2 Teams and roles

The different developers are involved in rather separate packages/subprojects and the most active developers in each are the ones with most influence. There are about ten packages, which have a varying number of active members. Developers that contribute the most are the ones making and carrying out any eventual major decisions (such decisions are discussed in the mailing lists first). Even though there are no formal roles, there are perhaps two levels; the core team and regular team members. Members of the core team have access to the main

repository and may commit code changes directly, whereas changes from others must be reviewed and approved by a member of the core team.

There are no strictly defined teams in the development; some developers have been part of the development for quite some time and are very active, whereas others may only be active for a limited amount of time (such as MSc/PhD students). Although heavily involved scientists are influential in the project, they have not been appointed any specific tasks or area of responsibility in the project. A problem with having too clear roles is that people may have periods where they have to do other things than contribute to the project (such as educating, supervising students or write papers) or that developers are involved for a limited time period.

7.1.3.3 Development process

Regular software process activities, such as design, analysis and implementation, are usually performed rather simultaneously. Later in the development, when the task is close to completion, testing activities may commence. During development, the code changes are regularly committed to either a separate branch or integrated directly in the main code repository. There is a perhaps a vague iterative development model, but it is not at all formalized or deliberate. The time and effort put into the different activities may vary from developer to developer and, moreover, from task to task, depending on a series of factors.

7.1.3.4 Requirements and testing

Requirements are handled in Launchpad and in mailing lists. Every member of the project may create tasks and report bugs, and also prioritize these and assign these. If a task is inserted with wrongful information in terms of priority, status or other things, it will be discussed by the developers and possibly changed. There are very seldom any problems of this kind, as people only use Launchpad actively when they are comfortable with it. Launchpad is also used to connect blueprints (bugs/tasks) to a specific release of the software. There are no plenary activities involved in the handling of requirements, meaning that any elicitation, specification and prioritization of tasks are performed individually. The only prerequisite when adding a task is that the descriptions are sufficiently explanatory.

The FEniCS project has engaged a dedicated tester, who has the responsibility of keeping a *Buildbot* up to date. The Buildbot is executed automatically whenever a developer commits code to the repository (and also every night). The existing tests are primarily regression tests, which checks calculations, but there are also some unit tests (covering about 10-20% of the software). The focus of unit tests has become greater recently, perhaps as a result of an imminent version 1.0 release of the software.

7.1.3.5 Challenges

The interviewee did not identify any aspects which could/should be better handled organizationally. If there are things not matching the demands of the project, these are addressed sooner rather than later. Such issues may be discussed intensively in mailing lists until a conclusion is reached. Once the preferred solution has emerged, the execution of it is

usually carried out very quickly, perhaps within hours. Generally, the ones that want the change are the ones that carry it out.

7.1.4 Summary

7.1.4.1 Teams and roles

Most of the developers are on the same level, much like the peer-based model of open source software development. The developers choose how much time and effort they devote to the project at a given time. Although there are no projects leaders, there are some subtle, implicit project roles; developers heavily involved, with a lot of contributions to their name, are very influential. There is a “core” team whose members review code changes. Membership in the core team is granted based on a constant level of contributions to the project. This means that the people that contribute the most are most influential.

7.1.4.2 Development process

The development process in FEniCS has been formed over the course of several years. Even though the process is undefined, certain practices have been established. The approach is incremental, where additional parts of the software are added in increments. The development process applied in the FEniCS project does not match any specific model. It was hard to identify any detailed process activities (and hence transitions between these), as the development is very much based on personal initiative and commitment. This may imply that there are several, vastly different approaches to the development of the software. Although the commitment is primarily on the individual level, there are still some issues need to be thoroughly discussed and coordinated (either via mailing lists or informal meetings).

Coding is clearly the most emphasized and time-consuming phase/activity. It is difficult to precisely define tasks beforehand, as they are so closely connected to research. There are multiple aspects which may influence a specific task during both planning and coding, such as changes to the original requirements and technical difficulties. As these types of challenges arise often, there is no focus on elaborate task specification or estimation. Developing the project is very much like conducting research (i.e. the output is not necessarily known), which means that one has to accommodate changes in requirements.

7.1.4.3 Requirements and testing

The level of self-organization is apparent in all activities related to requirements handling. Although LaunchPad is used to coordinate tasks and to track their progress, the specification and definition of a task is performed individually by the developer requesting the functionality. Some very minor tasks are not necessarily added in LaunchPad. No uniform pattern, such as user stories, is used for specifying tasks.

The project has a dedicated tester, who keeps a Buildbot up date. This test suite checks that the system compiles and that there are no build errors, as well as running existing regression

tests. Apart from the Buildbot, and some unit tests covering about 10-20 % of the code, testing is not particularly prioritized and such activities are left to the individual developer.

7.1.4.4 Other aspects

The scientists involved in the project are not co-located, but spread across the globe. As with any project with distributed development, collaboration, coordination and communication are key aspects which need to be handled appropriately in order for the project to be effective and successful. LaunchPad is an important tool for coordinating blueprints and for short discussions related to these. Another channel of communication is open mailing lists, where all types of matters related to the project are discussed. Both of these are pretty accessible and match the needs the project members have for coordination of tasks, bugs and eventual issues.

7.1.4.5 Challenges

Obvious challenges for FEniCS relates to collaborating and coordinating the project work. Sometimes, problems may arise when people have different requirements (such as accuracy demands) related to the same functionality.

7.2 Dalton

Three interviews were conducted in the Dalton project. Two of the interviews were individual, whereas the last had two developers present during most of the interview session. All the interviewed scientists are or have been very active in the project.

7.2.1 First interview

7.2.1.1 About interview/interviewee

This developer has been one of the main contributors in the project throughout and has been a member ever since its beginning in the early eighties. The interview is or has been active in other scientific software projects as well, but the main effort is invested in Dalton. This developer is very influential in the development and is part of an informal group of three project leaders. This is role is more directed towards research than the development of Dalton.

7.2.1.2 Teams and roles

The project structure is basically based on a set of more or less separate and individual teams (either one scientist or a small group of scientists). Within a single team, there is a strong sense of collaboration, but the level of external coordination/collaboration varies somewhat. There are no defined roles, but there are some slight differences among the various participants. PhD and MSc students report to their supervisor and have more constraints in terms of what they develop. Although they are thoroughly guided and focused on a particular part of the software, there is no strict delegation of tasks.

The customer and the developer is in most cases the same person. The Dalton project has slightly below 2000 users, which occasionally make requests or suggestions. There is, however, no guarantee that a request will be implemented. It is important that the task or request is of interest to a scientist for the task to be tended to. There are no paying customers, and hence no real obligations connected with following up input from users.

7.2.1.3 Development process

There are no explicit activities or stages in the development. A culture has emerged over the course of the project, which makes it possible to identify certain aspects. It is perhaps a somewhat iterative approach, as there are frequent changes in the inherent requirements.

Nearly all activities are performed individually, perhaps with an exception of planning. Planning meetings are arranged when needed. There are also weekly meetings in the interviewee's research group. Dalton matters may be addressed and discussed here, but the meeting is more related to the group's research, and thus not directly connected to the Dalton project.

7.2.1.4 Requirements and testing

Requirements are not gathered or specified systematically. This has not been necessary as the project is small enough for the active participants to have some idea of what the other developers currently are implementing (at least which part of the system they are working on). Requirements in the project are very tightly connected to research and are hence prioritized on an individual basis. The tasks are often quite large – taking months or even years to complete, and are often focused on a delineated part of the code. Collaboration and coordination activities are performed when there are overlaps in terms of functionality or research interests. Such activities are however seldom needed.

Every developer has to make sure that existing tests are passed before submitting code changes or new code. As other aspects of the development, testing is primarily performed on an individual basis. Any defined guidelines regarding testing do not exist; the individual developer is responsible for writing tests to his/her own code. The testing activity may be viewed upon as a kind of "safety" mechanism, as it is the only way the developer can "protect" his implementation from being changed by other scientists.

7.2.1.5 Other aspects

The software consists of two rather separate parts. The first part is the original, large Dalton software and contains a lot of functionality, mainly developed in Fortran77 (and some parts in Fortran90). The second part is much newer and exclusively in Fortran90. The new code is much more powerful in terms of performance, but does not yet cover all the functionality of the old code. The old part does not handle parallelism as well as the new software part does.

In terms of code standards, it is required that the developers try to code in similar fashion to the already existing parts. This is the case for code design also. The conductance of activities

associated with both code design and code standards is left in the hands of each individual developer.

7.2.1.6 Challenges

There are some aspects of the development which at times may be challenging. There are some difficulties related to testing; it is quite hard to design good quality tests and to keep them updated. All possible uses of the software are also quite hard to imagine prior to release. Much of the functionality is rather explorative and is implemented with the aim of exploring science. In such cases, there is an uncertainty whether the code and functionality actually is correct.

Due to complex program structures and frequent restructuring and alterations by many scientists over more than 25 years, the older parts of the software have gotten quite hard to understand and maintain.

7.2.2 Second interview

7.2.2.1 About interview/interviewee

This interview was not individual; two scientists were present during most of the interview. I did not regard this as a problem, as any disagreements between them are reported. The two developers are both currently active developers in the Dalton project and devote much time to the project; they regard themselves as close to full-time developers. These developers work very closely together and are located in the same room.

7.2.2.2 Teams and roles

There are no specific roles in the project. Some people, such as prominent scientists who have contributed a lot to the project over the years, may perhaps be more influential than other, but this it is not formalized in any way. The commitment to the project is personal and there are no-one in charge of organizing the project. For students collaborating closely with their supervisor the situation may be somewhat different; in such cases the student assumes more of a trainee-role, with the supervisor as his direct superior.

The customer and developer is often the very same person. There may be exceptions here, such as the above-mentioned student/supervisor-relationship. Users of the software may also provide requests to the project. Suggestions from the users must be of scientific value in order to be prioritized by a member of the project. Research is the top motivation for creating new functionality, but the researchers want to get as much as possible in return for their development effort and collaboration with users may be one way of achieving this.

7.2.2.3 Development process

It is challenging to identify any activities in the applied development process, as most of the development is performed individually or within a closely connected research group. There is

little point in extensive planning, as the plans would have to be changed often. There are consequently few meetings which focus on code and project structure, but coordination efforts must occasionally be performed. Another factor related to planning is that geographical spread of the developers and research groups makes it impractical to have regular meetings.

7.2.2.4 Requirements and testing

The requirements are usually formed by research interests and tasks are therefore very closely related to research. Handling of the requirements is usually conducted individually, but all requirements are gathered occasionally. In such cases the various tasks have been defined, assigned and prioritized. This has been proven quite useful in order to make all parts of the system work together. However, it is most common with personal “todo”-lists. Tasks may be hard to understand fully to begin with; during the implementation phase additional requirements may be discovered and errors may be revealed. Due to this explorative nature of the implementation, requirements are seldom estimated. It is also many other factors affecting how much time and effort one is able to invest in the project.

There is a test suite which must be run successfully before the code can be committed to the repository. The test suite consists of regression tests. The developers do not know how much of the code is covered by these. The two interviewees are both proponents for more and better testing, as it is very hard to locate code errors. There has been some discussion about whether to introduce unit testing, which would have provided a more precise localization of such errors.

7.2.2.5 Other aspects

The code can be divided into two rather separate parts; one old part which originate from the eighties and a new part which were formed about three years ago. The interviewees primarily work towards the new part of the code. The new code is more based on modules, rather than the monolithic structure of the old one. The old part of the code is difficult to understand and maintain. It has been subject to massive changes over the years by multiple scientists, which have had different motivations, mindset and intentions. In total, the new part of the code is much more maintainable and changes are easier to deploy here.

There are no guidelines for code practices in the project, which means that the focus on code quality and code standards varies quite a lot. There are huge variations as to the training and experience of the different participants in the project and every developer has his/her style.

7.2.2.6 Challenges

There are aspects of the development which are challenging. One such aspect is the previously mentioned code quality and standards. To determine the result upfront is hard or impossible. This complicates the identification of errors; it is not easy to determine whether an error belongs to the algorithm/science to be implemented or whether it is a programming bug.

7.2.3 Third interview

7.2.3.1 About interview/interviewee

The third interviewee has been an important member of the project and has been one of the main coordinators in a few releases. This developer had experience with programming prior to joining the project. His experience is some commodore64 programming (hobby basis), as well as a few courses at the university. The courses were about scientific computing and focused on technical aspects.

7.2.3.2 Teams and roles

Again, the interviewee found it difficult to identify roles. Perhaps it makes more sense to identify roles when a release is imminent. Releases balance on whether the scientists manage to collaborate and that someone take responsibility, which have been problematic at times. A small board has been established, in order to address such aspects. It is unclear how active this board has been to make such tough decisions and follow up on these.

7.2.3.3 Development process

It is difficult to formalize the activities in the process and possible transitions between them. Activities such as planning, analysis, design and implementation are performed almost simultaneously and the transitions between them are floating. There is rarely a distinct analysis step before one proceeds with the actual implementation of the task. Occasionally, mathematical formulas provide a kind of upfront specification, but usually one has a rather vague idea about the specifics of the task; quite a lot of programming has to be done in order to obtain a full overview of the task.

As with roles, activities are more evident when a release is pending. This developer has been heavily involved in deployment activities in the project and has taken an active role in some of the releases. Two deadlines are stipulated for the release; a deadline for checking in new features, and a final deadline. The time between these two dates is used to review the code, integrate the different branches of the code repository and to correct bugs and errors.

7.2.3.4 Requirements and testing

There have been periods where requirements-oriented meetings have been arranged, but not in a systematic manner. Tasks are often individual and oriented towards a specific section of the code. No plenary activities are conducted in relation to specification and elicitation of requirements. Coordination problems are handled by communicating with scientists engaged in the particular part of the software. Communication is preferably performed face-to-face, or by email if the involved developers are not co-located.

Testing practices are not very systematic. Approximately ten years ago, the developers created a framework for testing, which made it easier to test the results of executing the software. This initiative has not been entirely successful; there has been occasions were tests

have failed for a long period of time before finally being corrected. Consequently, tests have not always been reliable and developers may have neglected to write new tests and keep tests updated in periods where the test suite could not be trusted. Guidelines for testing have been adequate; it is more a matter of following these.

There have been some discussions about whether to incorporate unit tests, due to the fact that bugs are hard to locate. It seems to be a general consensus to opt for unit tests, but it has not been introduced yet. Unit tests would probable catch bugs at an earlier stage, but it is difficult to design tests covering all intended, and actual, use of the software. The interviewee is also unsure whether eventual guidelines here would be followed in the project throughout.

7.2.3.5 Other aspects

When this interviewee entered the project, there was no source code revision control. People exchanged code via email and a few central project participants had the responsibility of integrating all code changes.

The interviewee points out that the Dalton software, as of three years ago, consists of two rather separate parts. One of the parts is much newer and faster. This new part does not have all the functionality found in the old part of the software. Most of the development nowadays is directed towards the new part. It is unclear if, when and how the two parts will be consolidated, as the different parts are fundamentally different in terms of code architecture and code design.

7.2.3.6 Challenges

The interviewee was able to identify a few challenging aspects in the development, in addition to the previously mentioned testing difficulties. It is generally hard to understand and maintain some of the old code, perhaps even to the extent that one is afraid of making changes in certain parts. Another point, which the developer knows all about, is how difficult it is to coordinate the effort being put into the project in order to release a new version, especially getting all the scientists to commit the deadline and integrating the different branches in the code repository.

7.2.4 Summary

7.2.4.1 Teams and roles

It was hard to identify any roles in the project. Supervisors to MSc and PhD students assume roles akin to project leader for their students, but normally the scientists do not assume any roles at all. A board has been established in the project, whose responsibility is to map out the general, future directions of the project, as well as making decisions on significant matters in the project. However, the board is more related to research than the software development of Dalton. There is also a kind of user role, for people having signed the license agreement.

Users may request functionality or suggest changes, but there are no guarantees that the suggestions will be tended to.

7.2.4.2 Development process

There is no explicit development model, but a culture has been established over the years providing guidelines for how to attend certain aspects of the development. Nevertheless, most of the development is performed individually. Aspects requiring collaboration, such as integrating code or planning releases are handled in an ad-hoc manner.

None of the interviewees were able to identify any transitions between the process activities as most of these (such as planning, specification, design, analysis, testing and most importantly coding) are conducted more or less simultaneously. The development bears certain resemblances to iterative development models. However, the activities in the Dalton development process are not formalized at all.

7.2.4.3 Requirements and testing

The developers are located at various research facilities, most of them in Scandinavia. There are occasional meetings where requirements and current focus of development is discussed, but such meetings are far apart. The nature of a task and the related science may also play a part; occasionally the developers have a clear idea, but most of the time the full requirements are not known until way into the implementation phase.

Requirements are handled individually and all the interviewed developers have their own, private “todo”-lists. The level of self-organization is high; tasks are both defined and chosen by the developers themselves. Development is often motivated by (personal) research needs. There is seldom any gathering of the requirements on a plenary level; this is only done when a release is imminent. As there are relatively few people involved in the project at a given time, the most active developers seem to have some idea of what other members currently are implementing.

There are some regression tests which must be passed before the developer may commit his/her code to the main repository. The interviewees differ somewhat in their views regarding unit tests; their absence was pointed out by some but others stated that they have written such tests. Perhaps some parts of the regression test suite target single functions in the code (and in that sense constitutes unit tests).

7.2.4.4 Other aspects

Perhaps due to the long life-span of the project, no tools for managing code were used initially; this was done by exchanging source code via email. As the number of code lines increased, the need for a source code revision tool became more and more apparent.

It was pointed out in all the interviews that the software consists of two rather separate code parts. The newest part has greater performance and is considered to be easier to maintain, but does not include as much functionality as the old part yet.

7.2.4.5 Challenges

There are some challenges in the project. Releasing new versions of the software may be problematic due to the amount of coordination required. It is hard to get all the scientists to deliver in time and deadlines have occasionally been postponed. It is also difficult to manage the code and to integrate all the different code branches. Requirements are sometimes impossible to stipulate before well into the implementation stage, as many tasks are explorative and the correct output may be hard, if not impossible, to predict. It was also problematic to establish proper testing routines for all project participants to follow.

7.3 Olga

In the third project I interviewed two developers. Each interview lasted approximately one hour. As both developers were familiar with agile methodologies, all practices in the agile mapping chart were thoroughly discussed in these two interviews.

7.3.1 First interview

7.3.1.1 About interview/interviewee

The first interviewee has been involved in the project for several years. The educational background is primarily theoretical physics, with little emphasis on software engineering apart from a subject on programming techniques. This developer has attended a few courses about project management and Scrum methodology, after joining the company. Most of the work time is dedicated to programming software in the different Scrum-based projects in the company.

7.3.1.2 Teams and roles

The participants in the projects belong to one of three departments: *Research and Development*, *Maintenance* or *GUI*. Some of the developers are more connected to research-oriented parts of the system. In addition, developers may occasionally take on a support role for a limited time period (perhaps 1-2 months). Apart from the department-based roles, the regular ones from Scrum are used; every project has its own Scrum Master and Product Owner. There is a clear customer role, and customers from many different segments. There is not that much direct communication between customers and developers; most of this communication goes via the Product Owner. An exception is when the developer is part of a support team; in such cases the customer/developer relationship is more evident. There is also a dedicated tester in the company.

7.3.1.3 Development process

The company is rather large with many employees, which means that the overall development is divided into many sub-projects. Most of these projects use Scrum or a subset of the

practices associated with Scrum. The applied process depends somewhat on the Scrum Master and/or Product Owner in the project. Every project has individual deadlines and use iterations/sprints, with lengths of two to four weeks. Scrum has been used for a few years in the company and the choice and preference of adopting Scrum was probably motivated by the growth of the company and the management's need to retain control of the overall Olga project as it grew. Other factors may also have influenced the decision. Even though Scrum is used, there are only two yearly releases of the main software product.

7.3.1.4 Requirements and testing

A tracking system is used to handle requirements and tasks. Tasks have to be defined in this system in order for a developer to be able to commit the corresponding code changes. The developers try to estimate tasks to the best of their abilities and break them down to more manageable pieces/sub-tasks. The techniques used for performing task estimation vary somewhat from project to project; some use planning poker extensively, while others rely on expert estimation. The latter is used when a limited number of team members possess detailed knowledge of the task.

Every task must pass through the stages/steps of the tracking system. The steps for a task may differ slightly from other tasks, but common steps are specification and testing. However, they appreciate the fact that a straightforward path between the different steps is not always feasible; it is sometimes necessary to diverge from this, as changes may occur, or aspects may be uncovered, during later stages (such as implementation or testing) which affect the task(s).

The test suite consists of various use cases, which tests the results of executing the software. This test suite is run quite often, at least every night. They do also verify the results of the software with real, observed scientific data and make adjustments accordingly, as well as allowing selected customers to perform beta-testing on experimental versions of the software.

7.3.1.5 Challenges

There are challenges associated with establishing proper testing routines, but the overall project has shown a positive trend in recent years. Unit tests have only recently gained attention and no explicit routines for such tests have been established yet. Consequently, rather small parts of the code are currently being addressed by unit tests.

7.3.2 Second interview

7.3.2.1 About interview/interviewee

The second interviewee has a background from chemistry and begun with scientific programming as a graduate student. This developer has not received much formal education in computer science, apart from a few programming courses at the university or specific method courses (such as parallelization of software). The developer's software engineering knowledge and programming skills primarily come from practical experience.

7.3.2.2 Teams and roles

Each project has its own Product Owner and Scrum Master. The Product Owner often represents the customer in the project and works closely with the developers in the team. The company has also engaged consultants, which may have customer-type roles. The interviewee also points out that the department a developer belongs to (*Research and Development*, *Maintenance* or *GUI*) may affect his/her role in the overall project and what sort of tasks he/she usually is occupied with.

7.3.2.3 Development process

The applied process is first and foremost based on Scrum. The projects use the regular practices proposed by the technique, such as sprint backlog and sprint planning, review, retrospective and daily standup meetings. How well the actual applied practices conform to the ones prescribed by the Scrum methodology varies from Scrum Master to Scrum Master.

Scrum has been used at least for a few years. The sprints are from two to four weeks long. Sprint lengths vary from project to project, and perhaps even within a project. The team sizes do also vary, as do the total length of the different projects. Some projects may last for a single sprint only, while others have long life-cycles. The interviewee was uncertain about the underlying reasons for changing to Scrum in the first place, and is not sure how the applied process was prior to Scrum either.

7.3.2.4 Requirements and testing

To handle the requirements a tracking system is used. Issues created here have to pass through a number of steps before a developer may commit the change to the main repository. Some of the steps are connected to specification of the tasks, to the implementation and to testing. Tasks may be taken into sprints even though they are not completely specified or broken down, but they must have been subject to a certain level of specification beforehand. The specification from these early stages pretty much accounts for the technical documentation.

The test suite has traditionally consisted of several use cases. It is run daily, but it takes a long time to execute. The test suite covers approximately 40-50 percent of the production code. Unit tests have gained greater attention in the project lately. Producing unit tests for newly created code is pretty straightforward, but it is more difficult to create such tests when altering existing, previously untested, code; to create unit tests may exceed the time and effort available to make the requested code change in the first place. Unit tests do not cover a significant portion of the code yet, but such tests are created for much new code. There are, however, no formal demands that the developers *must* perform unit testing, which means that certain new code parts are not addressed by unit tests.

7.3.2.5 Other aspects

User documentation is often written near the end of a project, when the developers have a clear vision of how the end product turns out to be. Large parts of the code are rather

established and stable (as these have been used extensively by customers for many years). In the event that a developer wants to make changes in these parts, comprehensive research and clarification has to be made.

7.3.2.6 Challenges

There are a few aspects perceived as difficult or challenging. It is hard to imagine beforehand how things will look when completed. Consequently, estimating tasks and obtaining decent estimates may be very hard indeed. It is also hard to determine how to “attack” unclear tasks and they may grow more complex as one explores them. Most problems of the latter kind are caught during the various specification stages.

7.3.3 Summary

7.3.3.1 Teams and roles

In terms of roles, the developers belong to one of three main company departments; *Research and Development*, *Maintenance* and *GUI*. The GUI-developers seem to be engaged only in their own department, while other developers may engage in projects or activities officially belonging to either of the remaining departments. Occasionally, developers may also work with support-type tasks (viewed upon as a kind of maintenance effort). In the different subprojects, the regular Scrum roles are being used; every project has a dedicated Product Owner and a Scrum Master. The Product Owner and Scrum Master have their responsibilities and tasks to attend to, as do the regular team members/developers. There also exist other developer roles in the company, which do not specifically pertain to the Scrum methodology, such as a dedicated tester.

7.3.3.2 Development process

Due to the size of the company, the overall project (OLGA) is organized via multiple sub-projects of diverse size and duration. Most of these apply Scrum or at least many of the practices associated with Scrum. The regular Scrum meetings are usually arranged. The life-cycles of the projects are not synchronized, meaning that the projects have individual deadlines and backlogs. The iterations in the projects vary from two to four weeks. Releases of the complete software package are due approximately two times a year. Of course, as the projects differ in various aspects, the different Scrum practices are variably applied; some of the practices are followed throughout, while other practices may be absent or not carried out systematically, or at least not completely according to the Scrum methodology.

7.3.3.3 Requirements and testing

The requirements are handled in a tracking system, where the tasks must follow a series of states before ultimately being ready for the main repository. Scrum also provides some guidelines on how to deal with tasks, more specifically the sprint backlog and the sprint planning meetings. The tasks are planned, broken down to smaller, more manageable parts

and then estimated. Each project may choose its own way of estimating the tasks; some projects use planning poker, while others may use expert opinion.

A test suite covers close to fifty percent of the production code. The tests consist of use cases, investigating the results of the software. This test suite is run quite often, at least once a day. There are also some unit tests in the project, but as unit testing is a relatively new aspect of concern in the project, there are no specific or established guidelines related to creating and maintaining such tests. Thus, a limited part of the code is currently being addressed by unit tests. It is regarded as much easier to write unit tests when writing new code, compared to creating or updating them for already existing code. As parts of the software are scientifically explorative, the output cannot always be verified until one possesses real, observed data. Comparisons between the results from the software and the scientific data are conducted on a regular basis. Beta-testing of experimental versions of the software may also occur.

7.3.3.4 Challenges

There are a few challenges associated with testing. It is difficult to establish proper testing routines to be performed systematically in the project. Although a practice like for instance unit testing is regarded as important, not all new code are being addressed by such tests.

Whether the output matches real data is hard to assess prior to collecting the scientific, observed data. This uncertainty of desired results/functionality also complicates estimation activities. It is hard to obtain decent estimates because the developers may only have vague ideas of the task at hand, which means that a task's complexity may increase drastically – even after specification stages or implementation activities.

7.4 Agile practices in the projects

In this section the presence of agile practices in the three case study projects are investigated. The agile practices can be found in table 4, and detailed descriptions of each have been given in section 4.5.1. All 35 agile practices are reviewed one by one for each project. This section does not review each individual developer's perception of all practices, but sums up the combined impressions from each of the three projects. Eventual similar practices, but not completely alike the agile counterpart, are discussed. Disagreements between the developers are considered in the evaluation and practices hard to determine are accentuated. The complete mapping chart for all projects is presented in table 8 at the end of this section.

7.4.1 FEniCS

There were in fact a few different opinions among the developers, even though they all are (or have been) very heavily involved in the project.

7.4.1.1 Scrum practices

There is no Product Owner in the project. Priorities are handled individually or in mailing lists. The role of Scrum Master is not recognized either, at least not as a designated role. There are four yearly releases, which tend to follow releases of Ubuntu. Practice four is also not present; there is very little estimation of tasks altogether, and especially of plenary type. The project does not operate in time-boxed sprints or iterations. Also, the absence of practice 1 eliminates the possibility for practice 5 (any similar practices were not recognized either). All the first six practices are evaluated to no.

Daily stand-up meetings are not arranged. As the project development is distributed, open mailing lists is the primary communication channel in the project. There are occasionally some informal “face-to-face” meetings where such issues may be discussed. All the interviewees found the next practice to be present; the developers freely choose (and even define) their own tasks. The remaining Scrum practices, practice 9-12, are not present in the project. The project members do not measure the progress. There are no meetings for presenting new functionality, but things may be presented at yearly workshops, in lectures of various kinds or by email (in the open mailing lists). However, such presentations focus on the results obtained from the software. Retrospective meetings are not held. Issues impeding the development are discussed (on mailing lists or informal meetings) and eventual agreed-upon changes are carried out immediately. In terms of increments and release planning, there are seldom detailed planning farther ahead than the next release.

7.4.1.2 XP practices

Tasks are not specified as user stories; the primary focus is to provide enough information and the pattern or wording of the specification is of lesser importance. The following practice is also not present; the developers have their own offices and some of the members of the project are even located other places in the world. The practice of keeping a sustainable pace is of limited interest to the project, as the project involvement varies quite a lot (sometimes the developers are engaged with other activities, forcing them to contribute less to the project in certain periods). Measuring project velocity has not been considered. The developers choose themselves how much and in what way they contribute; the primary goal of the software development is to perform science. Developers in the project tend to focus on small parts of the code, (perhaps with the exception of some members of the core teams which go through a lot of the code). Thus, there is no focus on *moving people around* (to make developers work with different parts of the code).

The project does not have a clear customer/developer relationship. As personal motivation is key whenever a developer chooses and implements a task (meaning that the developer is almost always a customer/user of the software he/she currently develops), the practice stating that *the customer is always available* is evaluated to yes. The next practice is also evaluated to yes; the developers have established a code standard in the project. There exists a document describing how code should be formatted and designed. The interviewees regarded that code consistency is important, and that people generally follow the guidelines.

It is often hard to design unit tests prior to implementation. Unit tests are always written after the function is finished, if such tests are written at all. The interviewees have somewhat different views on pair programming; one interviewee engage in pair programming very rarely, where another may practice pair programming extensively in limited time periods. Although pair programming occasionally is practiced, it is undoubtedly more common that the scientists write code individually. Consequently, a limited part of the code is pair programmed. The code is also integrated individually, from the different developers' personal computers; they do not have a dedicated integration computer. Code changes are integrated as often as possible.

Officially, the code belongs to the members of the core team. The developers in this team are the only ones able to commit code directly to the main repository. This also means that other members' code changes have to go through them before these changes may become a part of the software. The developers try to retain a simple design, focusing on the task at hand. Any formalized system metaphor is not present in the project. The next practice is also absent, as there are no explicit design sessions. Such issues are primarily discussed either internally or in mailing lists.

Spike solutions may be created occasionally, but generally not. If such actions actually are carried out, the motivation is more likely performance issues rather than risk factors. Functionality is sometimes added early (for instance as dummy functions). In terms of refactoring, the interviewees had somewhat varying experiences; one of them refactored almost every time he encountered code of poor quality, whereas other developers may be more wary of making extensive changes. The developers are not afraid to refactor in order to improve the quality of the code, but it depends on the time available.

The coverage of unit tests is approximately between 10 and 20 percent. One of the interviewees mentioned that unit tests only recently became a focus in the project. All the tests that do exist must be passed for the code to be released. It is also a bit different opinions about whether tests actually are created when new bugs are encountered. One interviewee thinks that tests that address the programming bug will be created half the time, while another thinks that it will be done every time. Acceptance testing is not a distinct step in the development. The previously-mentioned Buildbot represents the user acceptance testing in the project. The test suite is executed regularly and the results are accessible online.

7.4.2 Dalton

There were some different opinions among the developers, but the consensus of whether the different agile practices were followed was for the most part unanimous.

7.4.2.1 Scrum practices

One interviewee point out that he sees some similarities between the Product Owner role and the status of developers who have been part of the development for a long time, but there is

broad agreement that such a role is not formalized. The Scrum Master role is not used either. There are no sprints (and no sprint backlogs) in the project; they do not operate on that time scale. Estimation of tasks is not usually conducted, as they feel the tasks are rather large and the effort required to complete them are unforeseeable. This would make eventual estimates very uncertain. Practice 5 and 6 are not present. The project members have in fact discussed whether to arrange daily stand-up meetings, but they have not made a decision regarding this yet (such meetings would probably concerned research as well, not just the development of Dalton).

The only Scrum practice evaluated to yes by the interviewees is number 8. Research is the top incentive for implementing new tasks or changing existing functionality. All developers choose what they implement themselves and they usually define their own tasks as well. How to implement the task is entirely up to each developer.

The developers do not keep track of their progress. It is also unusual that they present the software in any plenary fashion; it may occasionally be done by a lecture or during workshops (which are arranged 1-2 times a year), but the most common way of presenting new parts of the software (or results from the software) is by means of scientific articles. It became apparent during the interviews that process improvement actions in fact have been carried out in the course of the project. These improvements may be technical, such as introducing new code revision tools, or more process-based, such as establishing routines for testing or stricter guidelines for code standard (although there are some disagreements as to how well these guidelines and routines are followed). However, they do not arrange regular meetings where such issues are addressed. Planning increments is not performed in any systematic fashion; such meetings are arranged when needed.

7.4.2.2 XP practices

There is no need for very detailed or uniform task specification, as the developers work with delineated parts of the software and almost exclusively define and specify their own task. User stories are thus not used. The whole team is not situated in an open workspace environment, but it is not uncommon that small groups of developers are. Sustainable pace is rather irrelevant for the project, as Dalton is not a project engaging full-time developers (although some of the interviewees view themselves as close to full-time developers). The interviewees did not see any need for measuring the velocity of the project, as research is the most important factor. All interviewees point out that *moving people around (*i.e. get developers to work with different parts of the code) is not important, because one is generally focusing on a delineated part of the code.

Determination of whether practice 18 (*the customer is always available*) is followed or not depends on the definition of *customer*. One option is to define the customer and the developer as the very same person. The other existing customer/developer relationship is the one where users request functionality that could be of interest to the scientists. I find it most appropriate to use the first option, as the latter constitute a very marginal part of the development. Hence, the practice is evaluated to present. For practice 19 there were some disagreements among the

interviewees. It is certain that there exist structural guidelines the developers are encouraged to follow. Whether code in fact is written according to the guidelines is harder to determine. The developers have different opinions; some thinks they should have stricter routines to ensure that these guidelines are followed, whereas one developer thinks that proper reprisals actually are put forth if guidelines are not followed. It seems like the guidelines are not necessarily followed by all the project participants, meaning that the practice is probably not applied in the project.

The tests in the project are regression tests. Some of them are probably targeting specific functions in the code and is then essentially unit tests. However, none of the developers wrote tests prior to implementation. The interviewees have different opinions and experiences regarding pair programming. The most common way to code is individually, but developers may engage in pair programming occasionally. The next few practices concerns integration of code. As expected, one usually integrates code individually, as pair programming is more of an exception than a rule. There is not a dedicated computer for integrating. The only integration related practice whose evaluation is somewhat unclear is *Integrate Often*; the developers commit code (i.e. upload it to a branch in the code repository) quite often, but the main branch is not consolidated very often. Hence I find it most appropriate to evaluate the practice to not present.

Practice 25 is hard to determine. Although there is a collective "spirit", where all developers are authors of the software, the developers focus their effort on delineated parts of the code. People feel ownership to their part of the code. Because of this and the fact that some parts of the code is old and difficult to change, it is probable that the criteria (that all developers are able to contribute to main parts of the software) for the practice is not met. In terms of design the interviewees try to think a little bit ahead and make the code somewhat flexible. There was no evidence at all indicating the presence of the *Choose a system metaphor* (27) or the *Use CRC cards during design sessions* (28) practices.

It happens occasionally that the developers create small programs (spike solutions) to examine different solutions are available, but the primary motivation for doing so is performance, not to reduce risk. Functionality is sometimes added early (but in such cases, to a separate branch). Refactoring is done when needed and such efforts are affected by available time and developer discipline; sometimes one encounter parts of the code where one simply have to clean up a bit, while other times one might take a few shortcuts in order to get the code to work.

It is a bit uncertain whether the regression tests encompass unit tests. Nevertheless, there are parts of the code which are not being addressed by tests. The code must pass all existing tests before it can be released. These tests address various calculation and checks if the results are correct. When bugs are discovered, these are addressed more or less immediately. The majority of the interviewees think that in such cases the test suite will be modified, either by changing the existing tests or adding new ones. One of the developers points out that this is probably not done systematically by all the project's members, whereas the other interviewees

think that this is done consistently (at least when they themselves find bugs). Acceptance testing is not an aspect of particular interest in the project. The users of the software may provide feedback or report bugs when encountering them, but acceptance testing is not a formalized aspect or activity in the development process.

7.4.3 Olga

Scrum was used in many of the subprojects of Olga. Consequently, there were no considerable disagreements with respect to the Scrum practices. For the XP practices, the assessment was not as clear, but the interviewees agreed for the most part.

7.4.3.1 Scrum practices

Most of the Scrum practices are naturally present, as the Olga project explicitly uses this process methodology. The eight first practices are indeed present and performed consistently. All the projects do have a Product Owner and a Scrum Master. They also operate in iterations/sprints. The lengths of the sprints are between two and four weeks. There are however certain exceptions to the Scrum-based approach. Firstly, not all projects use all elements from Scrum if they do not find it necessary; for instance a project with only two developers do not necessarily have the same formalized manner of arranging sprint planning meetings as a larger project group. Secondly, planning poker is not the only estimation approach in the projects; sometimes expert opinion is used. Thirdly, daily stand-up meetings are not always possible, which means that projects may arrange such meetings every two days instead of every day.

Practice 9, concerning creation of burndown charts, is not applied thoroughly, but present in some of the projects. It is usual that the projects have review meetings, or demo meetings as they are referred to by the interviewees, each sprint. Retrospective meetings are also arranged in most projects. Whether release planning is present to a sufficient degree is hard to determine, but they seldom have future release plans beyond the next iteration. Consequently, the practice is evaluated to not present.

7.4.3.2 XP practices

User stories are heavily used and most tasks conform to the proposed pattern. Developers are generally not situated in an open workspace; some of them are, while most have their own offices. It is hard to determine whether practice 15 (*set a sustainable pace*) is present. There is not a lot of overtime for the developers, but often a bit more than forty hours/week is required in order to complete the project on time. This inclination indicates that the practice is not present in the project. The velocity has been measured in a few projects, but there have also been occasions where it has not. The practice is then evaluated as not present, as it is not applied systematically.

Moving people around is not necessarily a focus or priority in the project, but the different developers feel that they can contribute in most parts of the system. They are not limited to

focus their attention on delineated aspects or code packages. The customer is not always directly available, but the Product Owner, representing the voice of the customer, is. There are some established guidelines related to code standards, which the developers try to follow.

Unit tests are rarely written prior to implementation. In the very few cases where this has been done, it was done for new code. If one has to alter existing code, unit tests are added, if at all, after the code is modified. Pair programming is a very seldom practice in the projects, which means that both practice 20 and 21 can be evaluated to not present. Integration of code is done as often as possible. However, depending on the size and complexity of the task, it may take a while before something is operational. It is more important that the committed code is consistent and functioning than that code is integrated often. The developers integrate code from their individual computers; a dedicated integration computer has not been set up.

In terms of using a simple design, the two interviewees had somewhat different opinions. One of them claimed that they try to best of their ability to keep things separate, and if such efforts are successful they focus on the requirements as they are today. The other interviewee said that flexibility is not a major concern, but it is an aspect worth considering and that they try to facilitate future adjustments and changes (at least to a certain degree). This disagreement shows that there is no explicit focus on either adding flexibility or keeping the design as simple as possible. Consequently, the practice is evaluated to not present. The next two practices are also not present. No system metaphor is established. No specific design sessions are arranged. If meetings were such aspects were on the agenda, other design approaches than CRC cards were used.

The interviewees did not recognize spike solutions and found the practice to be not present. The next practice, however, is present; no functionality will be officially released unless it is finished and operational. The interviewees are not afraid to refactor the code, but whether they actually do refactor depends on how much time they have available and how extensive the change.

Not all of the production code has unit tests, not even all new code. Existing unit tests must be passed for the code to be released. The unit tests only cover limited parts of the production code. Tests are not necessarily created every time new bugs are encountered. Whether tests will be written or not depends on the time available and how difficult it is to create and update such tests. Even though customers are actively involved in testing the product, it is not an explicit phase in the project. There is then some level of acceptance testing but the results of such testing do not get published.

7.4.4 Summary of agile practices

The agile mapping chart of all three projects is presented below. Most of the practices were not present at all in case 1 and 2, whereas case 3 incorporated many agile practices.

	Projects		
#	*FEniCS*	*Dalton*	*Olga*
1	No	No	Yes
2	No	No	Yes
3	No	No	Yes
4	No	No	Yes
5	No	No	Yes
6	No	No	Yes
7	No	No	Yes
8	Yes	Yes	Yes
9	No	No	No
10	No	No	Yes
11	No	No	Yes
12	No	No	No
13	No	No	Yes
14	No	No	No
15	No	No	No
16	No	No	No
17	No	No	Yes
18	Yes	Yes	Yes
19	Yes	No	Yes
20	No	No	No
21	No	No	No
22	No	No	No
23	Yes	No	Yes
24	No	No	No
25	No	No	Yes
26	Yes	No	No
27	No	No	No
28	No	No	No
29	No	No	No
30	No	Yes	Yes
31	Yes	No	No
32	No	No	No
33	Yes	Yes	Yes
34	No	Yes	No
35	Yes	No	No

Table 8: Agile practices in the case study projects

8 Discussion

In this chapter, the results from the literature review and the case study are discussed in light of the two research questions. The first research question concerned the actual application of agile practices in scientific software projects and is primarily addressed by the case study. Consequently, the case study projects will be reviewed first, followed by the projects from the literature review. Differences and similarities between the projects, with regards to specific practices and the inherent characteristics of the projects will be subjects to thorough discussion.

Research question 2 concerned the effects of using agile practices in scientific software projects; the discussion will thus basically be based on the results obtained in the systematic literature review, in addition to the third project in the case study, which used several Scrum and XP practices. The other projects investigated in the case study will also be considered, especially when looking at the effects and whether any differences in testing and requirements activities were apparent. The end of the chapter is devoted to summarizing the results and comparing them with prior research on scientific software.

8.1 Presence of agile practices

As seen in table 8, some agile practices were present in the projects investigated in the case study, although many of them were not explicitly assigned to be used in the project. The practices which culminated over years of development did indeed resemble corresponding agile practices.

8.1.1 Scrum practices

The first twelve elements in the agile mapping chart are Scrum practices. Only one of the case study projects incorporated nearly all of these practices, whereas the other two only used one, namely number eight; *team members volunteer for tasks (self organizing team)*. There were often very individual drivers for change in the projects, and seeing as the customer is often the developer himself/herself, the handling of requirements is very much up to the individual developer. Olga differs from the other two cases in that respect, perhaps because it is a commercial project. Changes in requirements are more connected to customer demands. Responding to change is nevertheless very important for all the case study projects.

Retrospective meetings were only arranged by Olga, but all projects displayed a certain degree of adaptability when aspects or routines emerged as inadequate. In Dalton and FEniCS, there have been some elements of process improvements, including testing practices, review of code and tool usage. Alterations to the project and development have been initiated by key developers and changes to the process/project have been carried out more or less immediately. Such process improvement actions are not formalized in these two projects.

Estimation was not pointed out as a problem by FeniCS or Dalton developers (that is, it was recognized as a challenging aspect, but estimating tasks is of very little importance in the projects). This differs clearly from how Olga handles this activity, estimating all tasks using planning poker or expert-opinion.

FEniCS and Dalton do not have as defined a set of roles as prescribed by Scrum, while Olga uses both the Scrum Master and Product Owner roles. Olga is also the only project to arrange regular Scrum meetings, such as planning, review, daily stand-up and retrospective meetings. In the other two projects, similar topics are primarily discussed in mailing lists (or informal meetings if the developers are co-located). There is no tradition for presenting the software in FEniCS and Dalton. Contrary to Olga, FEniCS and Dalton do not have any established iterations in their development. However, the processes in these projects seem to have certain elements in common with iterative/incremental development approaches.

For the projects investigated in the literature review, it was far easier to determine which practices were present than identifying those that were not. The level of detail varied from article to article, and, although the processes were described and reported, it was generally not easy to rule out any practices based on the presented evidence.

Although the projects examined mostly used XP as their point of reference, some of the Scrum practices could be determined. Four of the twelve practices were in fact very common. Nearly all the projects used short sprints/iterations in their development processes, and every project used some form of release planning. Additionally, as is also the case with the projects in the case study, most of the projects' teams were self-organizing. The eventual presence of the remaining eight practices was, for the most part, hard to confirm. This may be due to the fact that the practice was not applied, but could also be caused by the article's focus. Team roles were not especially accentuated in most of the articles and it was thus difficult to determine whether the Scrum roles (or similar ones) were followed.

8.1.2 XP practices

Two of the XP practices were present in all the projects – the two practices being number 18, *The customer is always available*, and number 33, *All code must pass all unit tests before it can be released*. The customer role is somewhat different in the non-commercial projects, although Dalton has a customer segment via the issued licenses. In both these projects, the customer is usually the developer himself/herself and most commonly the developers define and prioritize their own tasks individually. It is unlikely that a scientist will dedicate time to a task if he/she does not have any personal research interest in the functionality. With regards to testing, all interviewees agree that the existing tests must be passed in order for the code to be released, but the unit tests only cover minor parts of the source code (if they exist at all).

User stories are only written in Olga, not in the other two projects. This may again be due to the fact that requirements are highly personal and individual in FEniCS and Dalton. The same trend is evident as to the estimation activities observed; in the non-commercial projects the

use of standardized tasks is perhaps limited, as these are rarely refined or estimated in any way, while developers in Olga perform such actions for every task. The same factors may also be the reasons why the practice of *moving people around* is absent in Dalton/FEniCS, but present in Olga. Olga has stronger traditions for involving developers in several parts of the code, indicating that people can be "moved around" and that tasks need to be more refined in order to enable most developers to attend to a given task. Code standards are regarded as important throughout and there are guidelines for this in all the projects. It is unknown how accessible these guidelines are in the Dalton project, and to what extent they are followed by all participating scientists.

The developers in the projects integrate their code changes rather often, preferably as soon as the code is working. Dalton developers seem to work towards separate branches. These branches are not fully integrated as often as in the other cases and there have been some problems consolidating the branches. The developers undoubtedly feel that the software is a collective effort, but in the Dalton and FEniCS projects the developers are so heavily involved in delineated parts of the software that they perhaps feel a special ownership to this part of the code. Simplicity in design is followed by the FEniCS developers. All the other interviewees said they tried to think a little bit ahead and facilitate eventual changes in the future, whereas FEniCS prioritized solving the task at hand and did not focus too much on flexibility.

All developers are careful about the code they integrate to the main branch. Dalton and Olga developers never commit code that is not fully functional. In the FEniCS project, non-functioning code occasionally finds its way into the main repository (for instance dummy functions). There are also some practices performed in just one of the three projects. The FEniCS developers found practice 34 (*refactor whenever and wherever possible*), to be a fitting description of how they approach code in need of refactoring. In the other two projects, there are certain code parts regarded as established and stable, which the developers approach very carefully indeed. Many factors affect refactoring issues; the time available perhaps being the most significant, but individual developer discipline and familiarity with the "poor code" also plays a part. User acceptance testing is performed every night for the FEniCS project and the score is published online. All the projects try to create tests when bugs are found, but only Dalton does so for every bug encountered.

For most of the developers in projects of this type, it made little sense to try to assess what is denoted by *sustainable pace*, as they are not full-time developers. For the Olga project, on the other hand, it makes sense to talk about the pace of development, but the interviewees regarded the workload of the project to be slightly more than "40 hours a week" on average and even somewhat higher when releases are imminent. Coding of unit tests before implementation are seldom done, but some parts of the development may be characterized as test driven (for instance that the developer create a prototype or demo prior to coding). Pair programming have been used in limited periods by some of the developers from the Dalton project, and to a lesser degree in FEniCS, but it is not a common practice in either. Pair programming is very rarely used in Olga. Spike solutions are however variably used and most of the interviewees pointed out that they did occasionally do so, but the reasons were to

increase performance, not to reduce risk. All the projects have been around for quite some time and have only recently begun to appreciate the advantages of unit tests. In FEniCS and Olga most of the newly created code has unit tests. There is however some exceptions, especially when changing or improving already existing code (which do not have unit tests). Developers usually do not have enough time available to create a full test suite for such code modifications.

Practices 14, 16, 22, 24, 27 and 28 were not even remotely applied or relevant. Neither of the projects used open work spaces, and they did see limited use for this in the daily development. Measuring project velocity is of limited interest in the non-commercial projects and was not systematically done in Olga. As pair programming were not used very often, the practice of having only one pair of programmers integrate at a time is even less relevant. The developers are not restricted from integrating their code and do so as often as they please. A dedicated integration computer appeared as a “strange idea” for most of the interviewees, and they did not see the point of going through extensive procedures to commit their code. The same may be said about the practice of choosing and establishing a system metaphor. CRC cards were perhaps not appropriate for all the projects investigated (especially not when dealing with old Fortran or C code) and using such cards in design sessions were not recognized at all. In fact, the projects did not have any specific design sessions with the whole team participating.

Some of the practices not applied by the projects in the case study were however present in the projects from the literature review. The first in that category is user stories, which were used by almost all literature review projects, but only used by Olga in the case study. This is a clear indication that user stories will probably not be written unless it is an agreed upon standard for specification of tasks. Another widespread practice for the literature review projects was to *use collective ownership*, again only shared with Olga. To *refactor whenever and wherever possible* were deemed as not present in all case study projects apart from FEniCS, but was generally done by the projects from the literature review.

Providing the team with *open work spaces* and *measuring project velocity* was not focus in any of the case study projects. The situation for the literature review projects however, is quite the contrary; both of these practices were among the most common. Out of the ten projects, only one did not measure the velocity (two of the articles did not report enough evidence to evaluate whether it was done). Another aspect where the trends from the literature review and the case study differs is for estimation activities. Although there was no specific inclination to the use of planning poker (for the projects in the literature review), most of the projects employed certain estimation activities. The team members from Olga were the only ones that estimated tasks on a detailed level in the case study; the other two projects had a general lack of estimation altogether.

The mapping chart from the literature review and the case study seem to conform very well for a few practices. This is especially the case for practice 18, which denotes that *the customer always is available*; this is enforced in every single project investigated in the thesis. As previously mentioned, the definition of the customer term is perhaps somewhat different in

scientific software as opposed to regular software engineering, but the practice is nevertheless applied throughout. *Integrate often* is also performed in almost all the projects investigated. Just a single practice could be evaluated to not present in most of the projects in the literature review, namely *all production code is pair programmed*. This was done consequently in one of the examined projects, but only performed sporadically in others, which corroborate the trends about pair programming from the case study projects.

8.1.3 Summary

Both literature review and case study projects displayed the presence of some of the agile practices. With the exception of Olga, who explicitly uses Scrum, it was difficult to find positive evidence for the presence of most of the agile practices.

Some practices were present in several projects, such as practices 5 (*time-boxed sprints producing potentially shippable output*), 7 (*short daily meeting to resolve current issues*), 8 (*Team members volunteer for tasks (self-organizing teams)*), 12 (*Release planning to release product increments*), 13 (*User stories are written*), 14 (*Give the team a dedicated open work space*), 16 (*The Project Velocity is measured)*, 23 (*Integrate often*), 25 (*Use collective ownership*) and 31 (*Refactor whenever and wherever possible*). Practice 19 (*Code written to agreed standards*) was also quite common, as it was used in six of 13 projects, and its absence only evident in one project. There was also positive evidence of regarding practice 33 (*All code must pass all unit tests before it can be released*) in three projects, but no evidence in either direction could be found in the remaining ones. Thus, the literature review and the case study indicate that 13 out of the 35 agile practices were used in the projects investigated.

Practice 21 (*All production code is pair programmed*) on the other hand, was clearly not present in most of the projects. Other practices, which were either impossible to determine or not present, were practices 24 (*Set up a dedicated integration computer*), 28 (*Use CRC cards for design sessions*) and 29 (*Create spike solutions to reduce risk*).

For the remaining 18 practices the picture is not clear. Many of these practices were impossible to determine for the projects found in the literature review. Nevertheless, for each of the practices, both negative and positive evidence were found regarding their presence, but there were no evident trends emerging.

8.2 Impact on challenging aspects

Most of the projects in the literature review did use a number of agile practices and reported on good and adequately effective handling of testing and requirements. In fact, these very aspects were emphasized in some of the articles as being especially successful when using an agile development approach. Unfortunately, not all the studies reported clearly which specific agile practices were applied, and to an even lesser extent which were not. Therefore, it is difficult to assess which practices did affect testing and requirements handling and the extent

to which the reported positive effects can be attributed to the use of these practices. Below, the practices that were used extensively are discussed as to their potential impact of the perceived good requirements and testing handling in the projects under investigation. Projects from both the literature review and the case study are discussed.

8.2.1 Requirements

Nearly all the projects from the literature review did incorporate a few central agile practices, which very well may have an impact on requirements and handling of such in the development processes. Practices 5 (*Time-boxed sprints producing potentially shippable output*), 8 (*Team members volunteer for tasks (self-organizing teams)*) and 12 (*Release planning to release product increments*) are all fairly common and the presence of these may well facilitate dynamic requirements as tasks frequently are discussed or refined. Practice 13 (*User stories are written*), used in many projects, could also further promote deliberate handling and refinement of requirements activities.

From the case study, the Olga project aligns well with the above-mentioned observations. Most of the Scrum practices were used and they regarded requirements activities to be dynamic and proper for the nature of the project. As the company switched to Scrum only a few years ago, the Olga project could have provided valuable insight into the effects of introducing an agile approach into a scientific software project. Unfortunately, none of the interviewees could recall the development approach used prior to Scrum, or whether there were any significant changes in requirements/testing handling. Thus, the effects of introducing Scrum are hard to evaluate. It is however certain that the Olga developers found the Scrum-based approach to fit their needs, and it is probable that the corresponding Scrum activities play a part in the performance of requirements and testing activities.

It is somewhat difficult to include the two non-commercial projects from the case study in this discussion, as they used rather few of the agile practices related to requirements. It is, however, interesting to observe that these projects did not have any particular problems with requirements, even though they did not use the agile practices. This might be explained by the fact that the development is based on personal motivation and that the individuals define and write their own requirements.

8.2.2 Testing

Five of the 35 agile practices are directly related to testing activities in software development projects: practices 20 (*Code the unit test first*), 32 (*All code must have unit tests*), 33 (*All code must pass all unit tests before it can be released*), 34 (*When a bug is found tests are created*) and 35 (*Acceptance tests are run often and the score is published*). Generally, it was hard to determine whether these activities were applied in the projects from the literature review, but some of the practices, such as practice 31, were thoroughly used and sporadic positive evidence was observed for the remaining test-related practices. Negative evidence for some of

the practices was observed in the case study, but was not possible to elicit in the literature review. In projects incorporating one or more of the test-related agile practices, problems with testing were less frequently reported.

Project 2 from the literature review used no less than four of the test-related practices (nr. 20, 32, 34, 35) and reported that the agile approach to testing was a valuable asset, concerning both the testing of new functionality and regression testing of existing functionality. In the case study, both FEniCS and Dalton used two of the test-related practices.

FEniCS applied practice 33 and practice 35, and it became apparent during the interviews that testing was not regarded as a problematic aspect of the development. This may indicate that the presence of these two practices at least contributes somewhat to the situation, but it is hard to provide clear support for a causal relationship between the presence of these practices and the lack of testing problems in the FEniCS. There are especially two factors that might be even more significant to this comfortable situation. Firstly, testing activity is of no real focus to many of the developers, and surely much less important than coding (i.e. conducting science). Secondly, the project has a dedicated tester whose responsibility is to maintain an automatically-run test suite.

Similar to FEniCS, the Dalton project did not emphasize particular problems with testing. Two test-related practices were used in the project (33 and 34). In addition, at least one of the developers frequently used practice 32. Again, eventual causal relationship between the presence of the agile test-related practices and the lack of perceived problems with testing is hard to determine, but there is at least some indication that such a relation *may* exist.

Project 4 from the literature review had positive evidence of one of the test-related practices (nr. 32). The remaining four could not be determined. Despite displaying clear evidence of only one of the practices, this was the project that most explicitly reported positive effects on testing activities. One of the observed effects was more focus and more deliberate handling of testing (an activity not prioritized before using agile practices). Arguably, practice 32 has much stronger conditions than most of the other test-related practices. It is very likely that the presence of this specific practice would mean that unit tests became subjects of recurring focus. This could be an indication that the presence of practice 32 has a massive impact on the development process, and in particular testing activities therein. As it is unknown whether the other test-related practices actually were present in the project, the prior notion is hard to determine. Another reasons why this project is most eager to attribute good handling of testing to the agile development approach could be that the focus of the study was to explore agile methodologies and assess whether it matched the requirements of scientific research, and that the developers did not have any prior experience with agile methods.

9 Limitations of the thesis

In this section the limitations of the thesis, both regarding the systematic literature review and the case study and other parts, are discussed, and its validity threats and the actions undertaken to limit the effects of such threats are described. The literature review is prone to validity issues in its general execution and in many of its stages, such as choice of databases, design of search query, data extraction and of course analysis. Perhaps equally important, there are certain limitations in the studies indentified also. Limitations and threats to the case study relates to how it is conducted as well as to the relevance of the cases to the proposed research questions.

9.1 Literature review

The way the review was conducted may be subject to certain limitations, among them the design of the search queries that were executed in the databases. The choice of databases may also influence the set of obtained studies. In a number of stages in systematic reviews, participation from several researchers is recommended, in order to ensure proper handling of the overall study, the search queries, data extraction and quality assessment. When multiple researchers take part, data extraction can be done independently followed by comparisons and discussions of these individual results. The review in this thesis was primarily carried out individually. To prevent risks of single-reviewer my supervisors and additional researchers assessed central parts of the systematic review, to ensure that its activities were executed properly.

It is primarily limitations in the identified studies that limit the validity, relevance and significance of the findings in the review. Internal validity issues were most salient in [32; 37], and especially evident in [37] where they introduced several new elements besides XP. Only one study considered the long-term effects of applying agile methods [35], but it is hard to make generalized assumptions based on this single study, particularly when the study only examined one project. Other internal validity issues were present in [32; 36], as they did not use a defined agile methodology. It is difficult to know how much of the results can be attributed to agile practices in the processes, even after the agile mapping chart was applied.

For external validity, the results of these studies cannot automatically be transferred to the general scientific software community. The settings of the projects and the composition of the teams were not necessarily representative for scientific software development, as evident in the earlier scientific software development characterization. Although a considerable number of projects were investigated, nearly all of these were of small size (generally 2 – 5 participants). The characterization studies do not necessarily agree, regarding the size of teams, but it is not uncommon that there are a large number of people involved in a scientific software project [3; 4]. Larger team sizes may provide extra obstacles for such projects. None

of the literature review projects were distributed, which is common in scientific software development. Both of these two factors may complicate the application of agile practices.

Also related to external validity is the domain of the projects investigated. The domain of most of the projects was bioinformatics, which is tightly connected to general informatics and computer science. Bioinformaticians may have an adequate understanding of software engineering concepts incorporated in their formal education, since a common career path is to take a bachelor degree in computer science and then proceed with further education in bioinformatics. It may be that the increased level of software engineering knowledge makes scientists in such projects more prone to apply software engineering concepts and practices and to succeed when doing so.

Limitations to the reviewed papers also limit the possibility of drawing conclusions in the literature review. Efforts have been made to take this into account when reporting the findings. Additional limitations to the systematic review include *reliability* threats due to single-reviewer assessment, *publication bias* due to papers possibly being submitted and published more readily when they report positive findings, and *selection bias* due to reviewer reliability threats and search engine mechanics. Publication bias can be ameliorated to some extent by also including “gray literature” (technical reports, unpublished material etc.), but the cost of retrieving such literature might be high compared to the potential benefits of an eventual retrieval.

9.2 Case study

The case study relied solely on interviews as this was the only type of evidence available. It is recommended that other evidence sources are gathered as well, such as observations, to corroborate the information elicited from interviews. No other types of evidence were available consistently in all three projects. As pointed out by Yin [21], there are a few common risks associated with interviews as an evidence source. These risks are described below, along with the precautions taken to limit the risk factors.

1. *Bias due to poorly asked questions*

In order to ensure that the interview guide was as good as possible, it was reviewed by experienced scientists. However, regardless of the quality of the questions in the interview guide, there might be certain misunderstandings and misconceptions when using technical terms and concepts from software engineering of which the interviewees and interviewer do not necessarily share the same connotations. Also, something may be lost or misinterpreted when the interviewer explains such concepts, in layman’s terms, for the interviewees.

2. *Response Bias*

This issue is mostly related to the wording of the questions. This is for two primary reasons not a significant issue in the interviews performed in the case study. Firstly, the interview guide, containing the set of questions, is, as previously mentioned, approved by other

researchers. Secondly, the questions in the guide are relatively open and general. This means that the questions are not leading, and hence the probability that the responses are affected by the questions themselves is very minor indeed.

3. *Inaccuracies due to poor recall*

The fact that all interviews were recorded digitally pretty much eliminates the risk of losing important evidence due to recall. This does, in turn, ensure the accuracy of the reported results.

4. *Reflexivity – interviewee gives what interviewer wants to hear*

It is quite difficult to determine whether this risk was present during the interviews; I do think, however, that most of the interviewees had a very vague idea of which responses the interviewer wanted to hear.

An advantage pertaining to the risks is that there were in fact no preferred responses, causing risk factors 2 and 4 to be less important in the outset. An approach put forth in Yin's book is to "corroborate interview data with information from other sources". This was not done in this case study, as other potential sources of evidence were not available and hence not collected. Preventative actions were undertaken in order to reduce the effect of the common risk factors associated with relying upon interviews as primary evidence source.

There were also validity threats related to the execution of the case study. Firstly, the developers in all cases were located at the same place, whereas the overall development projects were distributed. The perception and results obtained in the interviews do therefore primarily represent how the development is carried out at one of (possibly) many locations. It is not unthinkable that developers located elsewhere have a similar approach to the development, but they may very well use a different type of development. Moreover, it is not certain that they agree with the agile mapping chart obtained in the interviews, due to the fact that there might be significant variations regarding applied agile practices between the different locations. Secondly, there were some differences in how the individual interviews were conducted. Where the interviewee had little or no prior knowledge about agile methodologies and only limited formal training in general software engineering, there was not enough time to go through every single one of the 35 agile practices.

9.3 General validity threats

Some of the practices in the agile mapping chart are perhaps far too specific (for instance *Use CRC cards for design sessions* as opposed to *Team members volunteer for tasks (self organizing team)*. Also, the choice of references for the agile methodologies Scrum and XP affects the number of practices included in the chart. As there are no *de facto* standard for neither of the methodologies, it is hard to know which reference is most "correct" or reliable. The reference for XP [7] was chosen due to its richness (i.e. completeness) of practices

associated with the methodology, which ultimately may have lead to the inclusion of certain too specific practices. It is of limited interest to the research questions posed in the thesis precisely how many practices were followed or not, but the projects in the case study and research objects in the literature review could then be assessed on a very detailed level. Nevertheless, the inclusion also meant that the interview agenda had to include practices not particularly relevant, which may have caused the abovementioned problems (that some of the practices only could be discussed very briefly or not at all). The semi-structured interview approach enabled rapid alteration to the interview agenda if this proved necessary, for instance by limiting the discussion about practices which were presumably (based on the first part of the interview) not even remotely applied, and thus focusing the second part of the interview on agile practices more relevant to the specific project.

10 Conclusions and future work

Agile practices are used both explicitly and unintentionally in scientific software projects. In the projects investigated in the case study, most of these practices were, in fact, not used at all or only used sporadically. A select few practices were present throughout and identified as applied by all interviewees. One of these was the practice of self-organizing teams. Another practice followed consequently was that all unit tests have to be passed in order for the code to be released. However, the code coverage of such tests was rather low.

The results indicated that most of the practices associated with agile methodologies are not followed by all scientific software projects. Only in the exceptional case of Olga, a commercial project, the deliberate decision was made to use agile practices associated with the Scrum methodology. In the other projects, the practices which could be identified as frequently used are the ones that naturally lend themselves to the specific context of scientific software projects, such as practices 8 (*Team members volunteer for tasks (self-organizing team)*), 18 (*The customer is always available*), 23 (*Integrate often*), 25 (*Use collective ownership*) and 31 (*Refactor whenever and wherever possible*). All of these practices correspond with the conditions under which scientists develop scientific software, while other practices like, for instance, practice 21 (*All production code is pair programmed*) do not conform to such conditions.

The conclusion to the first research question is that contemporary scientific software projects embrace the agile spirit with its focus on flexibility and communication, but is selective as to the use of specific agile practices. In addition, some of the more technology-oriented and meticulous practices might simply not be known to the scientists in such projects as they are not professional software developers.

The thesis has shown that there might be a potential causal relationship between the use of agile practices and proper handling of requirements and testing activities in scientific software development. None of the projects displayed any negative side effects of using agile practices. There are some validity issues, both internal and external ones, for both the case study and the literature review. In order to obtain more conclusive answers to the second research question more research on the matter is essential. Nevertheless, the results from the literature review, as well as the case study, are indeed promising. Thus, a preliminary conclusion may be that the agile approach can be of great value to scientific software development, especially for smaller-sized teams and projects.

Further studies into agile practices in scientific software projects have to be conducted in order to draw generally valid conclusions related to the effects of using such practices. Investigating projects of a certain size and in fields other than bioinformatics would be valuable enhancements to the evidence base obtained in the literature review. Perhaps the research questions have to be addressed by case studies having more directly relevant cases; using projects incorporating a larger amount of the agile practices, and focus on their requirements and testing activities. Observations may be a proper evidence source to align

with interviews. Additionally, controlled experiments could be used to assess the effect of using agile practices, if appropriate research objects (scientific software projects introducing agile practices in their development) can be obtained. Projects like Olga, which switched to an explicitly agile process model, might be opportune cases in such studies.

A potentially interesting future direction of research in the development of scientific software could be to investigate how project characteristics affect the applied development process. Whether, and in what way, these characteristics, such as the life-span of a project, choice of programming language etc., affect the agile tendencies may be especially accentuated. This study indicates that there might be major differences between commercial and non-commercial scientific software projects and how these two types handle various aspects of the development process. Further investigations into this could be conducted.

Another possible research area is the challenges found in scientific software development processes. The challenges under investigation in the second research question, concerning requirements and testing, are only two out of many. There are other more or less important challenges that also need to be addressed by research, such as code review, collaboration difficulties between scientists and software engineers, code design and code maintenance. This study may also have identified a few additional challenges of possible significance. One of these was the mismatch between different developers regarding the accuracy of the software. Another project experienced problems when numerous code branches had to be consolidated (which complicated the release of new versions of the software). In order to resolve these types of issues and to identify suitable improvement actions, more research into explicit challenges and their possible solutions must be carried out.

11 References

[1] Kelly, D.F. 2007. *A Software Chasm: Software Engineering and Scientific Computing.* IEEE Software, Vol. 24, No. 6, pp. 118-120.

[2] Carver, J.C., Kendall, R.P., Squires, S.E. and Post, D.E. 2007. *Software Development Environments for Scientific and Engineering Software: A Series of Case Studies.* In Proceedings of the 29th International Conference on Software Engineering (Minneapolis, MN, USA, May 20-26, 2007), pp. 550-559.

[3] Sanders, R. 2008. *The Development and Use of Scientific Software.* MSc thesis, Queen's University (Kingston, Ontario, Canada).

[4] Granados, A.F. 2000. *A "scientific" approach to software project management Part 2: Results of a survey of scientific computing.*In Proceedings of the 10th Annual Conference for Astronomical Data Analysis Software and Systems (Boston, MA, USA, Nov. 12-15, 2000), pp. 20.

[5] Blom, M. 2010. *Is Scrum and XP suitable for CSE Development?* In Proceedings of the International Conference on Computational Science (Amsterdam, Netherlands, May 31- June 2, 2010), pp.1505-1511.

[6] Cohn, M. 2009. *Succeeding with Agile: Software Development Using Scrum.* Boston, MA : Addison-Wesley Professional.

[7] Wells, D. 1999. *The Rules of Extreme Programming*. [Cited: January 19, 2011.] URL: http://www.extremeprogramming.org/rules.html

[8] Hannay, J.E., Langtangen, H.P., MacLeod, C., Pfahl, D., Singer, J. and Wilson, G. 2009. *How Do Scientists Develop and Use Scientific Software?* In Proceedings of the Second International Workshop on Software Engineering for Computational Science and Engineering (Vancouver, Canada, May 23, 2009), pp. 1-8

[9] Beck, K. 2001. *Manifesto for Agile Software Development*, [Cited: January 19, 2011.] URL: www.agilemanifesto.org

[10] Paasivaara, M., Surasiewicz, S., Lassenius, C. 2008. *Distributed Agile Development: Using Scrum in a Large Project: A Multiple Case Study.* Proceedings of the Fourth International Conference on Global Software Engineering(Bangalore, India, Aug. 17-20, 2008). pp. 87-95.

[11] Cadle, J. and Yeates, D. 2008. *Project Management for Information Systems.* 5th Edition. Edinburgh Gate : Pearson Education Limited, p. 85.

[12] Granados, A.F. 1999. *A "scientific" approach to software project management Part I: A survey of development methodologies in scientific computing.* In Proceedings of the 8th

Annual Conference for Astronomical Data Analysis Software and Systems (Hawaii, USA, Oct. 3-6, 1999), pp. 19.

[13] Segal, J. 2008. *Models of scientific software development.*In First International Workshop on Software Engineering in Computational Science and Engineering (Leipzip, Germany, May 13, 2008).

[14] Segal, J. 2007. *Some problems of professional end user developers.* In Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (Idaho, USA, Sept. 23-27, 2007). pp. 111-118.

[15] Segal, J. 2001. *Organisational learning and software process improvement: a case study.* Book chapter in K.D. Althoff, R.L. Feldmann and W. Muller. *Advances in learning software organizations.* Milton Keynes, UK : Springer, pp. 68-82.

[16] Segal, J. 2005. *When software engineers met research scientists: A case study.* Empircal Software Engineering, Vol. 10, No. 4, pp. 517-536.

[17] Sanders, R. and Kelly, D. 2008. *Dealing with Risk in Scientific Software Development.* IEEE Software, Vol. 25, No. 4, pp. 21-28.

[18] Decyk, V.K., Norton, C.D. and Gardner, H. 2007. *Why Fortran?* Computing in Science and Engineering, Vol. 9, No. 4, pp. 68-71.

[19] Dybå, T., Dingsøyr, T. and Hanssen, G. K. 2007. *Applying Systematic Reviews to Diverse Study Types: An Experience Report.* In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (Madrid, Spain, Sept. 20-21, 2007), pp. 225-234

[20] Kitchenham, B. 2004. *Procedures for Performing Systematic Reviews.* Joint Technical report - Computer Science Department, Keele University and National ICT Australia Ltd.

[21] Yin, R.K. 2003 *Case Study Research: Design and Methods.* Sage Publications, Vol. 5.

[22] Taromirad, M. and Ramsin, R. 2008. *An Appraisal of Existing Evaluation Frameworks for Agile Methodologies.* In Proceedings of the 15th IEEE International Conference and Workshop on Engineering of Computer-Based Systems(Belfast, Northern Ireland, March 31 - April 4, 2008). pp. 418-427.

[23] Taromirad, M. and Ramsin, R. 2008. *CEFAM: Comprehensive Evaluation Framework for Agile Methodologies.* In Proceedings of the 32nd Annual IEEE Software Engineering Workshop(Kassandra, Greece, Oct. 15-16, 2008), pp. 195-204.

[24] Lappo, P. and Andrew, H.C.T. 2004. *Assessing Agility.* In Proceedings of 5th International Conference on Extreme Programming and Agile Processes in Software Engineering (Garmisch-Partenkirchen, Germany, June 6-10, 2004), pp.331-338.

[25] Datta, S. 2006. *Agility Measurement Index – A Metric for the Crossroads.* In Proceedings of the 44th Annual Southeast Regional Conference (Melbourne, Florida, USA, March 10-12, 2006). pp. 271 - 273.

[26] Seuffert, M. Agile Karlskrona Test. [Cited: June 7, 2011.] URL: http://www.piratson.se/archive/Agile_Karlskrona_Test.pdf.

[27] Waters, K. 42 point test. [Cited: June 7, 2011.] URL: http://www.allaboutagile.com/how-agile-are-you-take-this-42-point-test/.

[28] Little, J. *The Nokia Test*. [Cited: June 7, 2011.] URL: http://agileconsortium.blogspot.com/2007/12/nokia-test.html.

[29] Bowers, A.N., Sangwan, R.S. and Neill, C.J. 2007. *Adoption of XP practices in the industry - a survey.* Software Process: Improvement and Practice, Vol. 12, No. 3, pp. 283-294.

[30] Mirakhorli, M., Rad, A.K., Aliee, F.S., Mirakborli, A. and Pazoki, M. 2008. *RDP Technique: Take a Different Look at XP for Adoption.* In Proceedings of the 19th Australian Software Engineering Conference (Perth, Australia, March 26-28, 2008), pp. 656-662.

[31] Straub, P. 2009. Wikipedia.org: *A sample burndown chart.* [Cited: March 2, 2011.] URL: http://en.wikipedia.org/wiki/File:SampleBurndownChart.png.

[32] Easterbrook, S.M. and Johns, T.C. 2009. *Engineering the Software for Understanding of Climate Change.* Computing in Science & Engineering, Vol. 11, No. 6, pp. 64-74.

[33] Crabtree, C.A., Koru, A.G., Seaman, C. and Erdogmus, H.. 2009. *An Emprical Characterization of Scientific Software Development Projects According to the Boehm and Turner Model: a Progress Report.* In Proceedings of the Software Engineering for Computational Science and Engineering (Vancouver, Canada, May 23, 2009), pp. 22-27.

[34] Mugridge, R. 2003. *Test Driven Development and the Scientific Method.* In Proceedings of the Agile Development Conference (Salt Lake City, UT, USA, June 25-28 2003)**,** pp. 47-52.

[35] Pitt-Francis, J., Bernabeu, M.O., Cooper, J., Garny, A., Momtahan, L., Osborne, J., Pathmanathan, P., Rodriguez, B., Whiteley, J.P. and Gavaghan, D.J. 2008. *Chaste: using agile programming techniques to develop computational biology software.* Philosophical Transactions of the Royal Society - Series A: Mathematical, Physical and Engineering Sciences, Vol. 366, No. 1878, pp. 3111-3136.

[36] Kane, D.W., Hohman, M.M., Cerami, E.G., McCormick, M.W., Kuhlmman, K.F. and Byrd, J.A. 2006. *Agile methods in biomedical software development: a multi-site experience report.* BMC Bioinformatics , Vol. 7, No. 273, pp. 1-12.

[37] Wood, W.A. and Kleb, W.L. 2003 *Exploring XP for Scientific Research.* IEEE Software, Vol. 20, No. 3, pp. 30-36.

[38] Kane, D. 2003. *Introducing agile development into bioinformatics: an experience report.* In Proceedings of the Agile Development Conference (Salt Lake City, UT, USA, June 25-28 2003)**,** pp. 132-139.